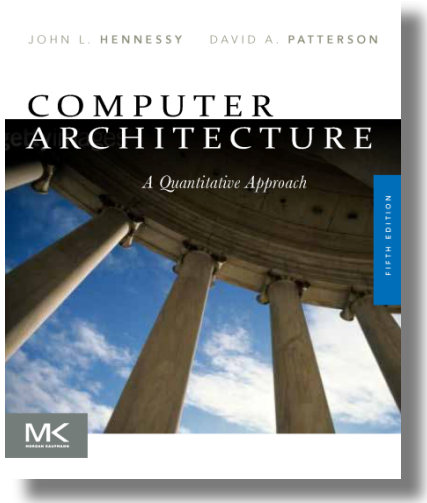


Instruction-Level Parallelism and Its Exploitation

Introduction

- Pipelining become universal technique in 1985
  - Overlaps execution of instructions
  - Exploits “Instruction Level Parallelism”
    - Overlapping of executions among instructions that are mutually independent
- Two main approaches to exploiting ILP:
  - Hardware-based dynamic approaches
    - Used in server and desktop processors
    - Not used as extensively in PMD processors
  - Compiler-based static approaches
    - Not as successful outside of scientific applications

Instruction-Level Parallelism

- When exploiting instruction-level parallelism, goal is to minimize CPI
  - Pipeline CPI =
    - Ideal pipeline CPI +
    - Structural stalls +
    - Data hazard stalls +
    - Control stalls
- Parallelism with basic block is limited
  - Typical size of basic block = 3-6 instructions
  - Must optimize across branches

Instruction-level Parallelism

- basic block: a straight line code sequence with no branches in except to the entry and no branches out except at the exit.
- For typical MIPS programs the average dynamic branch frequency is often between 15% and 25%, meaning
  - Basic block size *is between 4 and 7, and*
  - Exploitable parallelism in a basic block *is between 4 and 7*
- To achieve performance, we must exploit ILP across multiple basic blocks.

Instruction-level Parallelism

- Simplest and most common way to increase the ILP is to exploit parallelism among iterations of a loop
- Loop-Level Parallelism

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```

- Convert loop-level parallelism to ILP
  - **Loop unrolling:** static or dynamic
  - Use SIMD (vector processors and GPUs)

Data Dependence and Hazards

- To exploit ILP we must determine which instructions can be executed in parallel
- If two instructions are parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stall, assuming the pipeline has sufficient resources.
- Dependent instructions cannot be executed simultaneously

Data Dependence and Hazards

- Data dependences: There are three types of dependences
  - Data dependences (True data dependences)
  - Name dependences
  - Control dependences

Data Dependence

- Instruction **J** is data dependent on Instruction **I** if either of the following condition hold
  - Instruction **I** produces a result that may be used by instruction **J**

  - Instruction **J** is data dependent on instruction **K** and instruction **K** is data dependent on Instruction **I**

Data Dependence

- Caused by a "True Dependence" (compiler term)
- If true dependence caused a hazard in the pipeline, called a Read After Write (RAW) hazard
- If two instruction are dependent, they must execute in order and can not execute simultaneously or be completely overlapped.
- Dependencies implies that there could be a chain of one or more data hazards between two instruction

Data Dependence

- Dependencies are a property of programs
- Pipeline organization determines if dependence is detected and if it causes a stall

- Data dependence conveys 3 things:
  - Possibility of a hazard
  - Order in which results must be calculated
  - Upper bound on exploitable instruction level parallelism

- Dependencies that flow through memory locations are difficult to detect

Data Dependence

- Data dependence can limit the amount of ILP we can exploit.
- A dependence can be overcome in two ways:
  - Maintaining the dependence but avoiding the hazard
  - Eliminating the dependence by transforming the code
- Data value may flow between instruction either through register or memory locations.
- Detecting is straightforward when data flow occurs in register
- Dependencies that flow through memory locations are difficult to detect

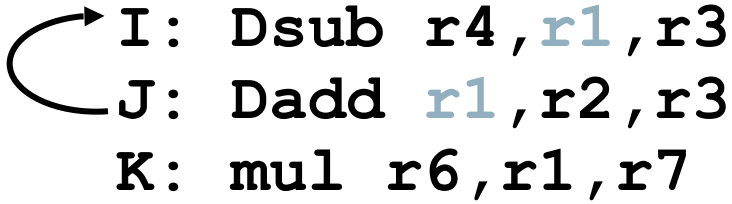
Name Dependence

- A Name dependence occurs when two instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name.

- Two types of name dependences between an instruction **I** that precedes instruction **J** in program order
  - Antidependence
  - Outputdependence

Anti-dependence

- An anitidependence between instruction **I** and instruction **J** occurs when instruction **J** writes a register or memory location **I** reads.

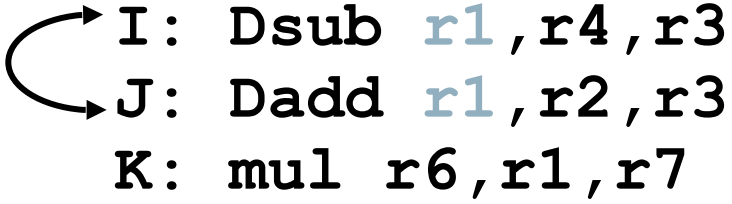


  - This results from reuse of the name "r1"

- Initial ordering (I before J) must be preserved

Output dependence

- An outputdependence occurs when instruction **I** and instruction **J** writes the same register or memory location.



  - This also results from the reuse of name "r1"

- Ordering must be preserved

Name Dependence

- Name dependence is not a true data dependence (*but is a problem when reordering instructions)* instructions involved in a name dependence can execute simultaneously or be ordered, if the name used in the instruction is changed so the instructions do not conflict

- To resolve, use renaming techniques (register renaming)

- Register renaming can be done statically by the compiler or dynamically by the hardware

Data Hazards

- Data Hazard exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.

- Must preserve the program order
  - The order that the instructions would execute in if executed sequentially one at a time as determined by the original source program

- Detecting and avoiding hazards ensures that the program order is preserved

Data Hazards

- Data Hazards are three types depending on the read and write access in the instruction

- Consider two instruction ***i*** and ***j***, with instruction ***i*** preceding ***j*** in the program order . The possible data hazards are:

  - Read after write (RAW):
    - ***j*** tries to read a source before ***i*** writes to it
  - Write after write (WAW)
    - ***j*** tries to write an operand before it is written by ***i***
  - Write after read (WAR)
    - ***j*** tries to write a destination before it is read by ***i***

Control Dependence

- A control dependences determines the ordering of an instruction *i* with respect to a branch instruction so that the instruction *i* is executed in the correct program order and only when it should be.

- Every instruction, except for those in the first basic block of the program is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

- Control dependencies must be preserved to preserve the program order.

Control Dependences

- Simplest example of a control dependence is the dependence of the statements in the then part of an if statement on the branch

- For example in the following code segment `S1` is control dependent on `p1`, and `S2` is control dependent on `p2` but not on `p1`.

```
if p1 {
   S1;
};
if p2 {
   S2;
}
```

Control Dependences

- There are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch:

```
if p1{                                  s1;
    s1;      cannot be changed to      if p1{
};                                      };
```

Control Dependences

- There are two constraints imposed by control dependences:

2. An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch:

```
s1;                          if p1{
if p1{      cannot be changed to      s1;
};                           };
```

Control Dependence

- In a simple pipeline control dependence is preserved by the following two properties:

  [1] Instruction execute in program order.

  [2] Detection of control or branch hazards ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

Control Dependences

- Control dependence is the fundamental performance limit.

- Control dependence need not always be preserved
  - Sometime we are willing to execute instructions that should not have been executed, thereby violating the control dependences, **ONLY IF** we can do so without affecting correctness of the program
- Control dependence is not the critical property that must be preserved.

Control Dependence Ignored

- Instead, TWO properties critical to program correctness – and are normally preserved by maintaining both data and control dependence – are
  - Exception behavior and
  - Data flow

Exception Behavior

- Preserving exception behavior means any changes in the ordering of instruction execution must not change how exceptions are raised in program
- Moreover, it should not give rise to no new exceptions in the program
- Example:

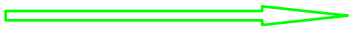
```
    DADDU    R2,R3,R4
    BEQZ     R2,L1
    LW       R1,0(R2)
L1:
```

- Problem with moving `LW` before `BEQZ`?

Instruction-Level Parallelism

- Examples:

```
  DADDU    R2,R3,R4
  BEQZ     R2,L1
LW         R1,0(R2)
L1:
```



```
  DADDU    R2,R3,R4
    LW     R1,0(R2)
  BEQZ     R2,L1
L1:
```

Data Flow

- The second property that is preserved by maintaining data dependence and control dependence is the Data Flow

- Data flow: actual flow of data values among instructions that produce results and those that consume them

- Branches make flow dynamic, since they allow the source of data for a given instruction to come from may points.
- An instruction may be data dependent on many of the predecessor.

Data Flow

- Program order is what determines which predecessor will actually deliver a data value to an instruction.

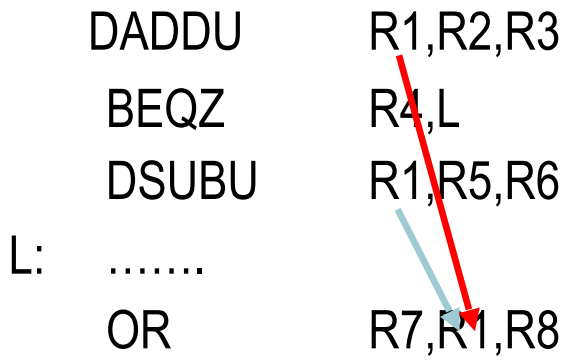
- Program order is ensured by the control dependence.
- Example:

```
DADDU    R1,R2,R3
BEQZ     R4,L
DSUBU    R1,R5,R6
L: …
OR       R7,R1,R8
```

- OR depends on DADDU or DSUBU?

Must preserve data flow on execution

Data Flow



By preserving control dependence of the **OR** on the branch, we prevent an illegal change to the data flow

- *Preserving data dependence alone is not sufficient for program correctness*
- *Instead, when the instruction executes, the data flow must be preserved.*
- *Speculation, which helps with the exception problem, can lessen impact of control dependence while still maintaining the data flow.*

Data Flow

- Sometimes violating the control dependence cannot affect either the exception behavior or the data flow

```
      DADDU    R1,R2,R3
      BEQZ     R12, SN
      DSUBU    R4,R5,R6
      DADDU    R5,R4,R9
SN:   OR       R7,R8,R9
```

The property of whether a value will be used by an upcoming instruction is called **liveness**

Is is possible to move DSUBU ?

- The move is okay if R4 is dead after SN, i.e. if R4 is unused after SN, and DSUBU can not generate an exception, since dataflow cannot be affected by this move.
  The code scheduling is sometimes called speculation.

Examples

- Example 1:

```
    DADDU R1,R2,R3
    BEQZ R4,L
    DSUBU R1,R1,R6
L:  …
    OR R7,R1,R8
```

- OR instruction dependent on DADDU and DSUBU

- Example 2:

```
    DADDU R1,R2,R3
    BEQZ R12,skip
    DSUBU R4,R5,R6
    DADDU R5,R4,R9
skip:
    OR R7,R8,R9
```

- Assume R4 isn't used after skip
  - Possible to move DSUBU before the branch

Control Dependence

- Control dependence is preserved by implementing control hazard detection that causes control stall.

- Control stall can be eliminated or reduced by a variety of hardware and software techniques.
  - Example: Delayed branch. Scheduling a delayed branch requires that the compiler preserve the data flow.

Eliminating Dependences

- **Name Dependences**

```
1    L5:   LD     F4, 0( R3)
2          ADDD   F6, F4, F2 ; data dependent on 1
3          SD     0( R3), F6 ; data dependent on 2
4          LD     F4,- 8( R3) ; name dependent on
                                1+2
5          ADDD   F6, F4, F2; data dependent on 4,
                  name dependent     on 2+3
6          SD     -8( R3), F6 ;data dependent on 5
7          SUBI   R1, R1,# 16
8          BNEZ   R1, L5  ; data dependent on 7
```

Eliminating Dependences

```
1    L5:   LD     F4, 0( R3)
2          ADDD   F6, F4, F2
3          SD     0( R3), F6      ; end of body 1
4          LD     F9,- 8( R3)
5          ADDD   F11, F9, F7
6          SD     -8( R3), F11    ; end of body 2
7          SUBI   R1, R1,# 16
8          BNEZ   R1, L5
```

Eliminating Dependences

- **Control dependences**

```
1       L5:     LD      F4, 0( R3)
2               ADDD    F6, F4, F2
3               SD      0( R3), F6
4               SUBI    R1, R1,# 8
5               BNEZ    R1, L5
6               LD      F4, 0( R3)      ; control dependent on 5
7               ADDD    F6, F4, F2      ; control dependent on 5
8               SD      0( R3), F6      ; control dependent on 5
9               SUBI    R1, R1,# 8      ; control dependent on 5
10              BNEZ    R1, L5          ; control dependent on 5
 exit:
```

Eliminating Dependences

- Control dependences -eliminate intermediate branches

1	L5: LD F4, 0(R3)	L5: LD F4, 0(R3)
2	ADDD F6, F4, F2	ADDD F6, F4, F2
3	SD 0(R3), F6	SD 0(R3), F6
4	SUBI R1, R1,# 8	SUBI R1, R1,# 8
5	**BNEZ R1, L5**	LD F4, 0(R3)
6	LD F4, 0(R3)	ADDD **F6**, F4, F2
7	ADDD F6, F4, F2	SD 0(R3), **F6**
8	SD 0(R3), F6	SUBI R1, R1,# 8
9	SUBI R1, R1,# 8	BNEZ R1, L5
10	BNEZ R1, L5	

Compiler Techniques for Exposing ILP

- Pipeline scheduling
  - Separate dependent instruction from the source instruction by the pipeline latency of the source instruction
- Example:

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Stalls

```
for (i=999; i>=0; i=i-1)
  x[i] = x[i] + s;
```

MIPS code for the above code

```
Loop:  L.D      F0,0(R1)     ; F0 = array element
       ADD.D    F4,F0,F2     ; add scalar in F2
       S.D      F4,0(R1)     ; store result
       DADDUI   R1,R1,#-8    ; decrement pointer 8 bytes
                             (per DW)
       BNE      R1,R2,Loop   ; branch R1 != R2
```

Pipeline Stalls

```
Loop:   L.D       F0,0(R1)        1
        stall                     2
        ADD.D F4,F0,F2            3
        stall                     4
        stall                     5
        S.D F4,0(R1)              6
        DADDUI R1,R1,#-8          7
        stall                     8
        BNE R1,R2,Loop            9
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Pipeline Scheduling

Scheduled code:

```
Loop:   L.D      F0,0(R1)
        DADDUI R1,R1,#-8
        ADD.D F4,F0,F2
        stall
        stall
        S.D F4,8(R1)
        BNE R1,R2,Loop
```

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling

- Loop unrolling: Increasing the number of instructions relative to the branch and overhead instructions is loop unrolling

- Unrolling replicates the loop body multiple times, adjusting the loop terminating code.

Loop Unrolling

- Loop unrolling
  - Unroll by a factor of 4 (assume # elements is divisible by 4)
  - Eliminate unnecessary instructions

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;drop DADDUI & BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;drop DADDUI & BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

- note: number of live registers vs. original loop
- Loop will take 27 CC

Loop Unrolling

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;
         DADDUI R1,R1,# -8;  drop BNE
        L.D F6,-8(R1)
        ADD.D F8,F6,F2
        S.D F8,-8(R1) ;
         DADDUI R1,R1,#- -8; drop BNE
        L.D F10,-16(R1)
        ADD.D F12,F10,F2
        S.D F12,-16(R1) ;
         DADDUI R1,R1,# -8;  drop BNE
        L.D F14,-24(R1)
        ADD.D F16,F14,F2
        S.D F16,-24(R1)
        DADDUI R1,R1,# -8
        BNE R1,R2,Loop
```

- note: number of live registers vs. original loop
- Loop will take 27 CC

Loop Unrolling

```
Loop:   L.D F0,0(R1)
        ADD.D F4,F0,F2
        S.D F4,0(R1) ;drop DADDUI & BNE
        L.D F0,-8(R1)
        ADD.D F4,F0,F2
        S.D F4,-8(R1) ;drop DADDUI & BNE
        L.D F0,-16(R1)
        ADD.D F4,F0,F2
        S.D F4,-16(R1) ;drop DADDUI & BNE
        L.D F0,-24(R1)
        ADD.D F4,F0,F2
        S.D F4,-24(R1)
        DADDUI R1,R1,#-32
        BNE R1,R2,Loop
```

- note: number of live registers vs. original loop
- Loop will take 27 CC

Loop Unrolling/Pipeline Scheduling

- Pipeline schedule the unrolled loop:

```
Loop:   L.D F0,0(R1)
        L.D F6,-8(R1)
        L.D F10,-16(R1)
        L.D F14,-24(R1)
        ADD.D F4,F0,F2
        ADD.D F8,F6,F2
        ADD.D F12,F10,F2
        ADD.D F16,F14,F2
        S.D F4,0(R1)
        S.D F8,-8(R1)
        DADDUI R1,R1,#-32
        S.D F12,16(R1)
        S.D F16,8(R1)
        BNE R1,R2,Loop
```

- Loop will take 14 CC

Strip Mining

- Unknown number of loop iterations?
  - Number of iterations = *n*
  - Goal: Unroll the loop to make *k* copies of the loop body
  - Generate pair of consecutive loops:
    - First executes *n* mod *k* times and has a body that is the original loop
    - Second is the unrolled body surrounded by an outer loop that iterates (*n* / *k*) times
    - "Strip mining"
- For large value of n, most of the execution time will be spent in the unrolled loop body

Loop-Level Parallelism

- Loop-level parallelism is normally analyzed at the source level or close to it
  - Finding parallelism involves recognizing structures such as loops, array references, and induction variable components .
- Loop-level analysis involves determining what dependences exist among the operands in a loop across the iteration of that loop
  - Determine whether data accesses in later iterations are dependent on data values produced in earlier iteration
  - Loop-carried dependence

Loop-Level Parallelism

```
for (i=1; i<=1000; i=i+1)
                x[i] = x[i] + s;
```

- Dependence between two uses of X[i]
- Dependence between successive uses of **i** in different iteration

Loop-Level Parallelism

```
for (i=1; i<=100; i=i+1) {
                A[i+1] = A[i] + C[i]; /*S1*/
                B[i+1] = B[i] + A[i+1]; /*S2*/
        }
```

- What are the data dependence among the statement S1 and S2 in the loop?

Loop-Level Parallelism

```
for (i=1; i<=1000; i=i+1) {
                A[i] = A[i] + B[i];       /*S1*/
                B[i+1] = C[i] + D[i];     /*S2*/
        }
```

- What are the data dependence between S1 and S2? Is this loop parallel? If not show how to make it parallel?

Loop-Level Parallelism

- A loop is parallel if it can be written without a cycle in the dependences
  - Absence of a cycle means that the dependences give a partial ordering on the statement

Loop-Level Parallelism

```
for (i=1; i<=100; i=i+1) {
                A[i] = A[i] + B[i];          /*S1*/
                B[i+1] = C[i] + D[i];        /*S2*/
        }
```

- There is no dependence between from S1 to S2
  - Interchanging the two statement will not affect the execution of S2
- On the first iteration of the loop, statement S1 depends on the value of B[1] computed prior to initiating the loop

Loop-Level Parallelism

```
for (i=1; i<=100; i=i+1) {
                A[i] = A[i] + B[i];          /*S1*/
                B[i+1] = C[i] + D[i];        /*S2*/
        }
```

```
A[1] = A[1] + B[1];
        for (i=1; i<= 99; i=i+1) {
                B[i+1] = C[i] + D[i];
                A[i+1] =  A[i+1] + B[i +1];
        }
B[101] = C[100] + D[100];
```

Loop-Level Parallelism

```
for (i=1; i<=100; i=i+1) {
                    Y[i] = X[i] / c;          /*S1*/
                    X[i] = X[i] + c;          /*S2*/
                    Z[i] = Y[i] + c;          /*S3*/
                    Y[i] = c – Y[i];          /*S4*/
               }
```

Find all dependences and eliminate o/p and antidependences by renaming.

Branch Prediction

- Control dependencies become the limiting factor to the amount of ILP we attempt to exploit
- More crucial to processor that tries to issue more than one instruction per clock cycle
  - Branches will arrive up to n times faster in an n-issue processor and providing stream to the processor will probably require that we predict the outcome of branches
  - Amdahl's law remind us that relative impact of control stall will be larger with the lower potential CPI in such machine
- The effectiveness of a branch prediction scheme depends not only on the accuracy but also on the cost of a branch when the prediction is correct and when incorrect

Branch Prediction

- Branch penalties depend
  - On the structure of the pipeline
  - Type of predictor
  - Strategies used for recovering from mis-prediction

Correlating Branch Predictors

- 2-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch

- It may be possible to improve the prediction accuracy if we also look at the recent behavior of other braches rather than just the branch we are trying to predict

Correlating Branch Predictors

```
If (aa==2)
        aa=0;
If (bb==2)
        bb=0;
If (aa!=bb){
```

Assign aa and bb to registers R1 and R2

```
    DADDUI   R3,R1,#—2
    BNEZ     R3,L1          ;branch b1  (aa!=2)
    DADD     R1,R0,R0       ;aa=0
L1: DADDUI   R3,R2,#—2
    BNEZ     R3,L2          ;branch b2  (bb!=2)
    DADD     R2,R0,R0       ;bb=0
L2: DSUBUI   R3,R1,R2       ;R3=aa-bb
    BEQZ     R3,L3          ;branch b3  (aa==bb)
```

- The behavior of branch b3 is correlated with the behavior of branches b1 and b2
  - If b1 and b2 both not taken then b3 will be taken
  - A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

- Branch predictors that use the behavior of other branches to make prediction are called correlating predictors or two-level predictors.

Correlating Branch Predictors

```
If (d==0)                    BNEZ    R1,L1          ;branch b1  (d!=0)
        d=1;                 DADDIU  R1,R0,#1       ;d==0, so d=1
If (d==1)                L1: DADDIU  R3,R1,#-1
Assign d to register R1      BNEZ    R3,L2          ;branch b2  (d!=1)
                             …
                             L2:
```

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	Yes	Not taken	1	Yes	Not taken
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

Correlating Branch Predictors

```
If (d==0)                    BNEZ    R1,L1          ;branch b1  (d!=0)
        d=1;                 DADDIU  R1,R0,#1       ;d==0, so d=1
If (d==1)                L1: DADDIU  R3,R1,#-1
Assign d to register R1      BNEZ    R3,L2          ;branch b2  (d!=1)
                             …
                             L2:
```

Behavior of a 1-bit Standard Predictor Initialized to Not Taken

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Correlating Branch Predictors

- The standard predictor mis-predicted all branches
- Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch.

- Ex. a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch.

Correlating Branch Predictors

- Consider a predictor that uses 1 bit of correlation.

- Every branch has two separate prediction bits
  - One prediction assuming that the last branch executed was not taken and the other if the last branch executed was taken
  - In general the last branch executed is not the same instruction as the branch being predicted

Correlating Branch Predictors

- A 1-bit correlation predictor uses two bits:
  - First bit being the prediction if the last branch in the program is not taken
  - Second bit being the prediction if the last branch in the program is taken

Prediction bits	Prediction if last branch not taken	Prediction if last branch taken
NT/NT	NT	NT
NT/T	NT	T
T/NT	T	NT
T/T	T	T

Correlating Branch Predictors

The Action of the 1-bit Predictor with 1-bit correlation, Initialized to Not Taken/Not Taken

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	**NT**/NT	**T**	T/NT	NT/**NT**	**T**	NT/T
0	T/**NT**	**NT**	T/NT	**NT**/T	**NT**	NT/T
2	**T**/NT	**T**	T/NT	NT/**T**	**T**	NT/T
0	T/**NT**	**NT**	T/NT	**NT**/T	**NT**	NT/T

```
If (d==0)
        d=1;
If (d==1)
Assign d to register R1
```

```
    BNEZ     R1,L1        ;branch b1   (d!=0)
    DADDIU   R1,R0,#1     ;d==0, so d=1
L1: DADDIU   R3,R1,#-1
    BNEZ     R3,L2        ;branch b2   (d!=1)
…
L2:
```

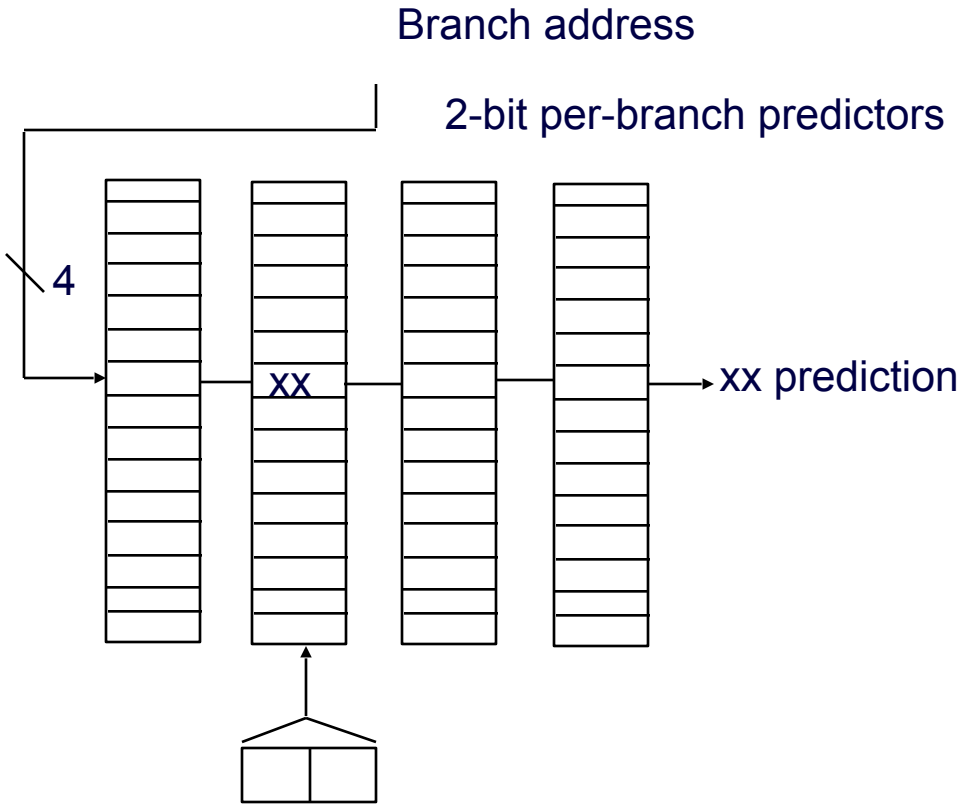
Correlating Branch Predictors

- With the 1-bit correlation predictor, also called a *(1,1) predictor*, the only mis-prediction is on the first iteration!

- In general case an (*m,n*) predictor uses the behavior of the last *m* branches to choose from 2^m branch predictors, each of which is an *n*-bit predictor for a single branch.

Correlating Branch Predictor

- Advantages
  - Higher prediction rate than 2 bit predictor
  - Require only trivial amount of additional hardware
- Simplicity of H/W comes from the from the following observation
  - The global history of the most recent *m* branches can be recorded in an *m*-bit shift register, where each bit records whether the branch was taken or not
  - Branch prediction buffer can then be indexed using a concatenation of the low-order bits from the branch addresses with *m*-bit global history


Branch address
2-bit per-branch predictors
4
xx
xx prediction
2-bit global branch history

The number of bits in an (m,n) predictor is:

- ***2^m*n *(number of prediction entries selected by the branch address)***
- A 2-bit predictor with no global history a (0,2) predictor

- How many bits are in the (0,2) branch predictor we examined earlier? How many bits are in the branch predictor shown in (2,2) branch predictor

- How many branch-selected entries are in a (2,2) predictor that has a total of 8K bits in the prediction buffer?

2-bit predictor with 4K entries and a (2,2) predictor with 1K entries

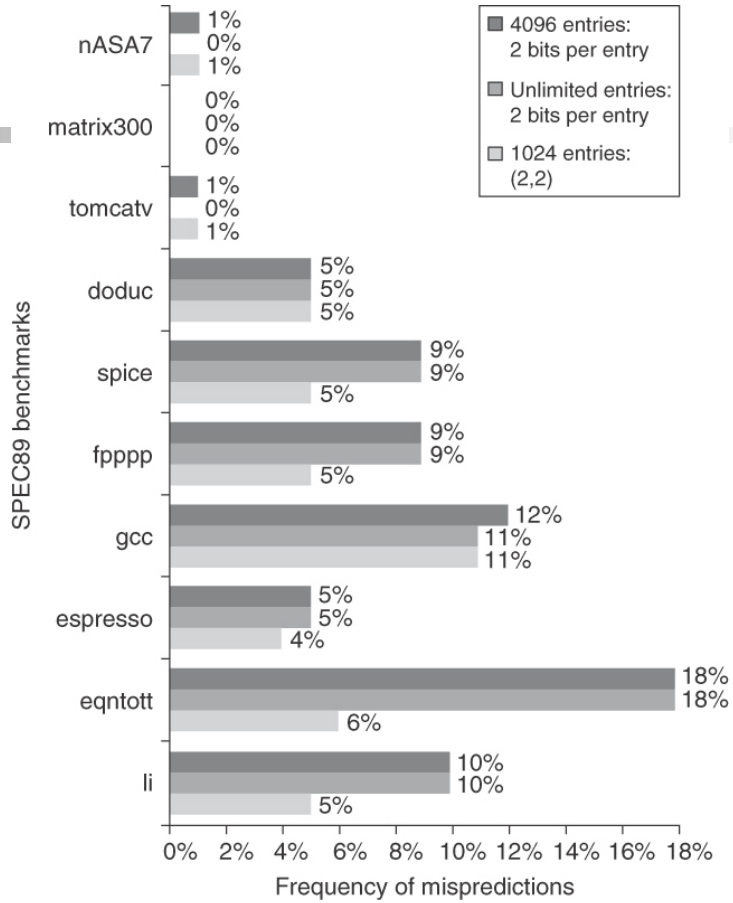


Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

Branch Prediction

- Basic 2-bit predictor:
  - For each branch:
    - Predict taken or not taken
    - If the prediction is wrong two consecutive times, change prediction
- Correlating predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes of preceding *n* branches
- Local predictor:
  - Multiple 2-bit predictors for each branch
  - One for each possible combination of outcomes for the last *n* occurrences of this branch
- Tournament predictor:
  - Combine correlating predictor with local predictor

Tournament predictor

- Uses multiple predictors
  - One based on global information
  - One based on local information
  - Combine them with a selector
- Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor was most effective in recent predictions.
- The saturation counter requires two mis-predictions before changing the identity of preferred counter

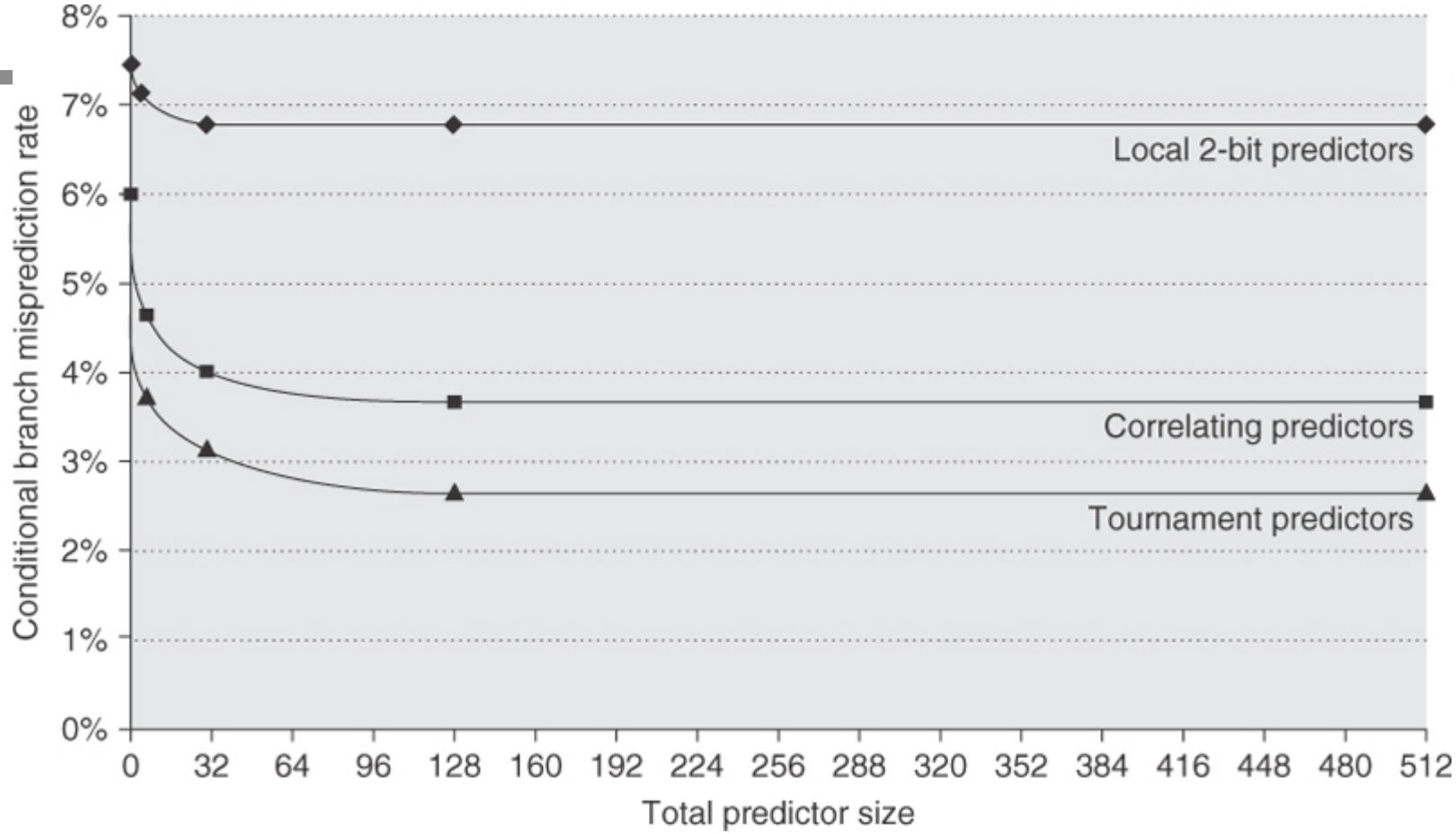


Figure 3.4 The misprediction rate for three different predictors on SPEC89 as the total number of bits is increased. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar behaviour, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

Intel Core i7 Branch Predictor

- i7 uses a two-level predictor
  - Smaller first-level predictor, designed to meet the cycle constraints of predicting a branch every clock cycle
  - A larger second-level as a backup
- Each predictor combine three different predictors
  - A simple two-bit predictor
  - A global history predictor
  - A loop exit predictor

Intel Core i7 Branch Predictor

- Loop exit predictor uses a counter to predict the exact number of taken branches for a branch that is detected as a loop

- In addition
  - A separate unit predicts target address for indirect branches
  - A stack to predict return address

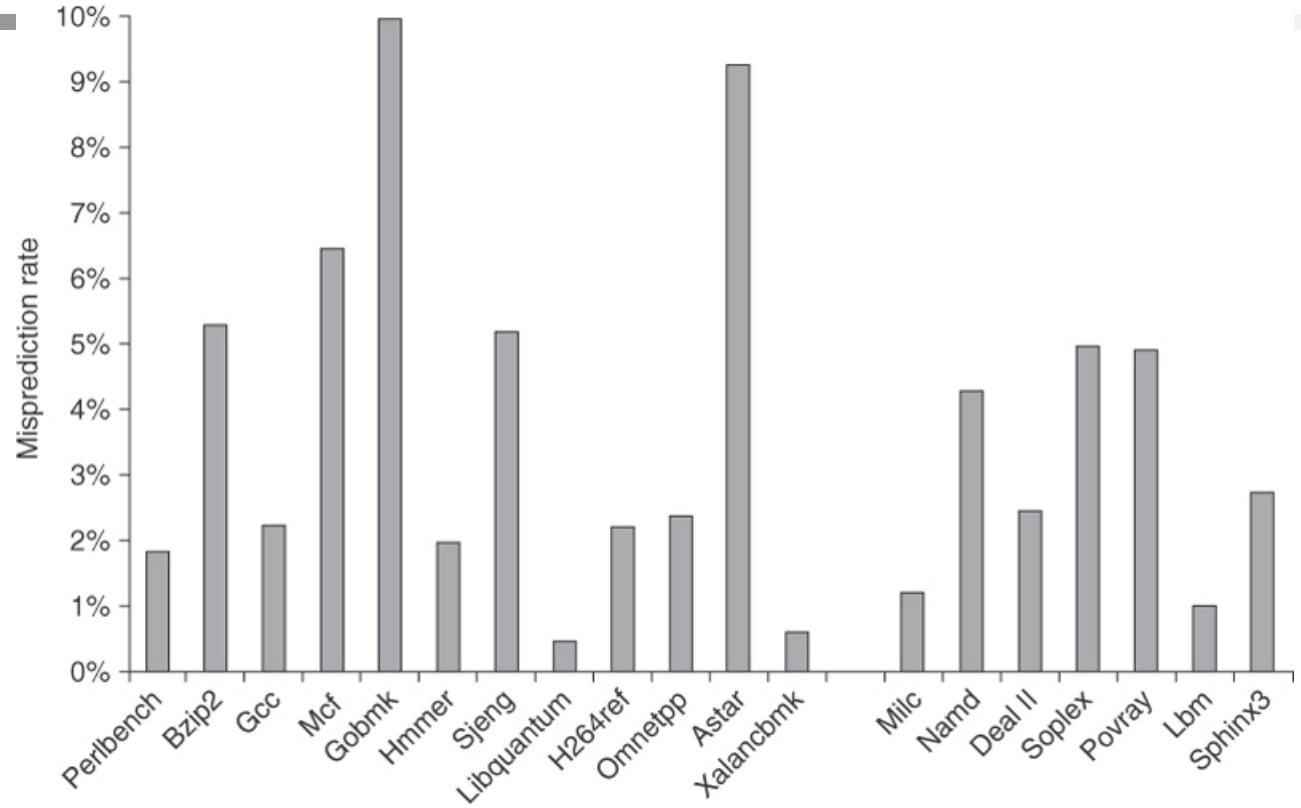


Figure 3.5 The misprediction rate for 19 of the SPECCPU2006 benchmarks versus the number of successfully retired branches is slightly higher on average for the integer benchmarks than for the FP (4% versus 3%). More importantly, it is much higher for a few benchmarks.

Dynamic Scheduling

- Rearrange order of instructions to reduce stalls while maintaining data flow and exception behavior
- Advantages:
  - Compiler doesn't need to have knowledge of microarchitecture
  - Handles cases where dependencies are unknown at compile time
  - Allow processor to tolerate unpedictable delays
- Disadvantage:
  - Substantial increase in hardware complexity
  - Complicates exceptions

Overcoming Data Hazard Dynamic Scheduling

- **Static Scheduling : If there is a hazard, stop the issue of the instruction and the ones that follow**
- **Dynamic Scheduling : Hardware rearranges the execution of instructions to reduce stalls while maintaining data flow and exception behavior.**
- **Advantages:**
  - **Allows code compiled for ONE pipeline to run on another**
  - **Handles cases when dependences are unknown at compile time (e.g. involve a memory reference)**
  - **Allows processor to tolerate unpredictable delays such cache miss**

Dynamic Scheduling

- Advantages of dynamic scheduling are gained at the cost of significant increase in hardware complexity
- Dynamically scheduled processor cannot change the data flow
  - It tries to avoid stalling when dependencies that could generate hazards
- Static pipeline scheduling by the compiler tries to minimize stalls by separating dependent instructions so that they will not lead to hazards.

Dynamic Scheduling: Idea

- Major limitation of pipeline:
  - **In-order instruction issue and execution**.
- Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instruction can proceed.

- In case multifunctional unit these unit remain idle.

Dynamic Scheduling

If instruction **j** depend on a long-running instruction **i**, currently in execution in the pipeline then all instruction after **j** must be stalled until **i** is finished and **j** can execute.

Dynamic Scheduling

```
DIV.D    F0,F2,F4
ADD.D    F10,F0,F8
SUB.D    F12,F8,F14
```

- Both structural and data hazards could be checked during instruction decode (ID)
- Check for structural hazard and wait for the absence of data hazard.
- Key idea: Allow instructions behind stall to proceed
- Enables out-of-order execution and allows out-of-order completion

Dynamic Scheduling

- Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exits in five- stage integer pipeline and its logical extension to an in-order floating point pipeline.

Dynamic Scheduling

```
DIV.D    F0, F2, F4
ADD.D    F6,F0,F8
SUB.D    F8,F10,F14
MUL.D    F6,F10,F8
```

- WAR
- WAW

- Both could be avoided by use of register renaming.

Out-of-order Completion

- Also creates major complication in handling exceptions.
- Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that exactly those exceptions that would arise if the program were executed in strict program order actually do arise.
- Exception is preserved by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed.

Out-of-order Completion

- Dynamically scheduled processor may generate imprecise exception

- An exception is imprecise if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order.

Out-of-order Completion

- Imprecise exceptions can occur because of two possibilities:
  1. The pipeline may have **already completed** instructions that are **later** in program order than the instruction causing the exception.
  2. The pipeline may have **not yet completed** some instructions that are **earlier** in program order than the instruction causing the exception.
- Imprecise exceptions make it difficult to restart execution after exception.

Dynamic Scheduling

- To allow out-of-order execution, ID stage of the simple 5-stage pipeline is split into TWO stages:
  - Issue—Decode instructions, check for structural hazards
  - Read operands—Wait until no data hazards, then read operands

Dynamic Scheduling

- IF stage precedes the Issue stage.
- Instruction are fetched either into a register or into a queue of pending instruction.
- Instructions are issued from the register or queue
- EX stage follows the read operand stage
- Execution may takes multiple clock cycle

Dynamic Scheduling

- Will distinguish when an instruction *begins execution* and when it *completes execution*; between these two times, the instruction is *in execution*
- Multiple instructions are in execution
- Requires multiple functional units, pipelined functional units, or both

Out-of-order execution

- In a dynamically scheduled pipeline, all instructions pass through issue stage in order (in-order issue)

- But they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order (out-of-order completion).
  - Scoreboardig: sufficient resources, no data dependences

  - Tomasulo's algorithm: tracks when operands are available to minimize RAW hazards, and introduces register renaming to minimize WAW and WAW hazards.

A Dynamic Scheduling Using: Tomasulo's Approach

- Allow the execution to proceed in the presence of dependencies was used in IBM 360/91 floating point unit.

- Invented by Robert Tomasulo:
  - Tracks when operands for instructions are available, to minimize RAW hazards, and introduces register renaming to minimize WAW and WAW hazards.

A Dynamic Scheduling Using: Tomasulo's Approach

- IBM 360/91 was completed before cache
- IBM's Goal: Achieve high FP performance from an instruction set and from compilers designed for the entire 360 computer family rather than from specialized compilers for high end processors.
- 360/91 architecture has only 4 double precision FP registers which prevented interesting compiler scheduling of operations
  - This led Tomasulo to try to figure out how to get more effective registers — renaming in hardware!

A Dynamic Scheduling Using: Tomasulo's Approach

- IBM 360/91 had long memory accesses and long FP delays.
- Why Study 1966 Computer?
- The descendants of this have flourished!
  - Alpha 21264, HP 8000, MIPS 10000, Pentium III, PowerPC 604, …

Dynamic Scheduling

- RAW hazards are avoided by executing instruction only when its operand are available

- WAR and WAW hazards are eliminated by register renaming
  - Renaming all destination registers, including those with a pending read or write for an earlier instructions, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.

Dynamic Scheduling

```
DIV.D    F0, F2, F4
ADD.D    F6, F0, F8
S.D      F6, 0( R1)
SUB.D    F8, F10, F14
MUL.D    F6, F10, F8
```

Assume there exist two temporary register S and T. Using S and T the above code sequence can be rewritten as the following

Dynamic Scheduling

DIV.D	F0, F2, F4
ADD.D	S, F0, F8
S.D	S, 0(R1)
SUB.D	T, F10, F14
MUL.D	F6, F10, T

Tomasulo

- Register renaming is provided by the Reservation Station
- Basic ideas
  - Reservation stations fetches and buffers the operands as soon as it is are available
  - Need to read from registers is eliminated
  - Pending instruction designate the reservation station that will provide their input
  - When successive writes to a register overlap in execution only the last one is actually used to update the register.

Tomasulo

- As instructions are issued the register specifiers for pending operands are renamed to names of Reservation Stations.
- Register renaming that avoids WAW and WAR

Tomasulo

The use of reservation station leads to two other important properties:

1. Hazard detection and execution control are distributed:

    Information held in the reservation stations at each functional unit determine when an instruction can begin execution at that unit.

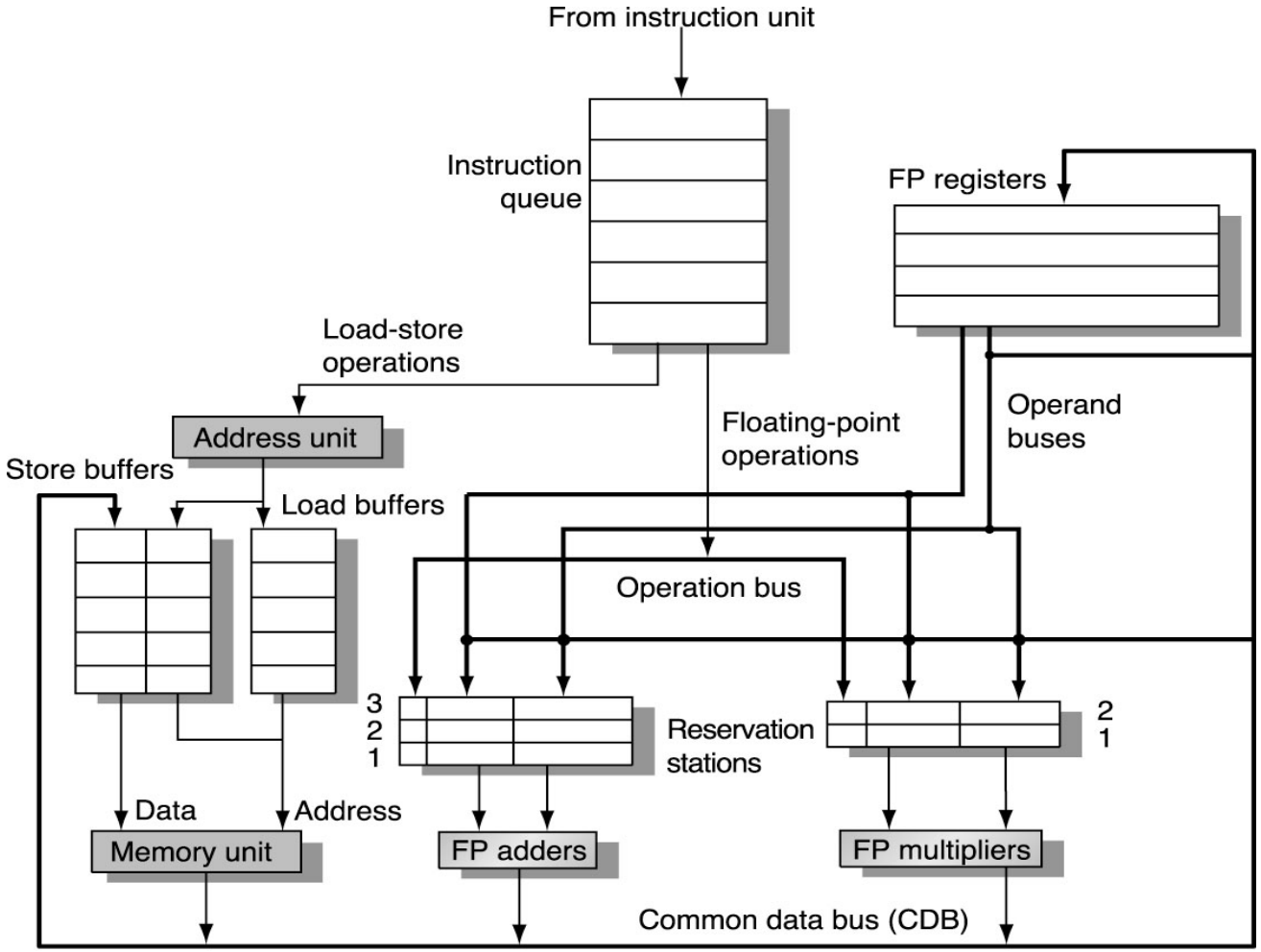
2.Results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers.

This bypassing is done with a common data bus called common data bus (CDB) in 360/91.

All units waiting for a result can be loaded simultaneously.

In pipeline with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

Basic structure of a MIPS FP unit using Tomasulo's algorithm



More Components

- Reservation Stations(RS) :
  - Hold instruction that has been issued and is waiting execution at the FU
  - Hold operand value for that instruction or the name of the RS that will produce the operand
- Load buffers :
  - Hold the component of EA until it is computed.
  - Track outstanding load awaiting on memory.
  - Hold the result of completed loads that are awaiting for he CDB.
- Store buffers : hold data/ address going to memory
  - Hold the component of EA until it is computed.
  - Hold the destination memory addresses of outstanding stores hat are waiting for the data value to store.
  - Hold the address and value to store until the memory unit is available.

- **Issue :**
  - get next instruction from the instruction queue, maintained in FIFO order
  - issue the instruction to the matching reservation station, if empty
  - send operands to the reservation station if they are currently in the register; if operands are not available, write the reservation station that will produce them
  - if there is no empty reservation stations then there is a structural hazard; instruction is stall until a reservation station / buffer is freed
  - This step renames registers, eliminating WAR and WAW

Steps of an Instruction

- Execute
  - If one or more operand is not available monitor the CDB while waiting for it to be computed
  - When an operand is available, put it placed in the corresponding reservation station
  - When all operands are available, the operation can be execute in the corresponding reservation station
  - By delaying until all operands are available, RAW hazards is avoided

Steps of an Instruction

- Execute
  - Independent functional units can begin executing in the same cycle
  - If two or more instruction becomes ready for a single FU then one is chosen to execute
  - FP reservation station can choose a instruction arbitrarily. However, load and store pose additional complication

Steps of an Instruction

- Execute:
  - Load and store requires two- step execution
  - First step: computes the effective address when base register is available and then the effective address is placed in the load or store buffer
  - Second step: actual memory access
  - Loads in the load buffer execute as soon as the memory unit is available
  - Store in the store buffer wait for the value to be stored before being sent to the memory
  - Load and Store are maintained in program order through effective address calculation

Steps of an Instruction

- **Execute:**

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in the program order have completed.

Branch prediction allow execution of instruction and not stall until it enters Write Result

- **WB :**

– Write result on CDB. From there, it goes to register and reservation stations/ store buffers waiting for the result.

– If store: write to memory

Steps of an Instruction

- **Write Result:**
  - Write result on CDB when available
  - From the CDB it goes to registers & reservation stations including store buffer waiting for the result.
  - Store write to memory: When both the address and data values are available, they are sent to the memory unit and the store completes.

- Data structure used to detect and eliminate hazards are to the reservation station, registers and load and store buffers.

  Each of them are attached with a tag

- They are names for extended set of virtual register used in renaming

- Tag field describes which reservation station contains the instruction that will produce a result needed as a source operand.

- Reservation station and buffers are like registers

- Once an instruction is waiting for an operand, it refers to the operand with the tag number of the reservation station or buffer t h a t will produce it
- Since there are more reservation station than architectural registers WAW and WAR hazards are eliminated by renaming results with RS

Tags in Tomasulo scheme refers to the buffers or unit that will produce a result.

Register names are discarded when a instruction issues to a reservation station.

Reservation Station Components

- Each reservation station has seven fields

  Op: Operation to perform on source operand S1 and S2

  Qj, Qk: Reservation stations that will produce the corresponding source

  - Qj, Qk = 0 : Indicates the source operand is already available or unnecessary

  Vj, Vk: Value of Source operands
  - For loads the V_k field is used to hold the offset field

Reservation Station Components

- Each reservation station has seven fields

  Busy: Indicates reservation station and its accompanying FU is busy

  A: Used to hold information for the memory address calculation for a load or store. Initially the immediate field of the instruction is stored. After address calculation the effective address is stored.

Reservation Station Components

Register file has field Q_i

Indicates which functional unit will write each register, if one exists. Blank or 0 when no pending instructions that will write that register.

Load and Store buffers have A field which hold the result of effective address once the first step of execution has been completed.

Tomasulo Eample

- Assumptions of execution times for different instruction types

LD	1 clock cycle
ADDD, SUBD	2 clock cycles
MULT	10 clock cycles
DVID	40 clock cycles

Tomasulo Example

Instruction stream

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2			
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

FU count down

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock: **0**

	F0	F2	F4	F6	F8	F10	F12	...
FU								

Clock cycle counter

Tomasulo Example Cycle 1

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1		
LD	F2	45+	R3			
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	*Address*
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
1	*FU*				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1		
LD	F2	45+	R3	2		
MULTD	F0	F2	F4			
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU		Load2		Load1					

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
3	FU Mult1	Load2		Load1					

- Note: registers names are removed ("renamed") in Reservation Stations; MULT issued
- Load1 completing; what is waiting for Load1?

Tomasulo Example Cycle 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
4	*FU*	Mult1	Load2		M(A1)	Add1				

- Load2 completing; what is waiting for Load2?

Tomasulo Example Cycle 5

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
5	FU	Mult1	M(A2)		M(A1)	Add1	Mult2			

- Timer starts down for Add1, Mult1

Tomasulo Example Cycle 6

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
6	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Issue ADDD here despite name dependency on F6?

Tomasulo Example Cycle 7

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
7	FU	Mult1	M(A2)		Add2	Add1	Mult2			

- Add1 (SUBD) completing; what is waiting for it?

Tomasulo Example Cycle 8

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

- Add2 (ADDD) completing; what is waiting for it?

Tomasulo Example Cycle 11

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
11	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Write result of ADDD here?
- All quick instructions complete in this cycle!

Tomasulo Example Cycle 12

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
15	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult1 (MULTD) completing; what is waiting for it?

Tomasulo Example Cycle 16

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
16	*FU*	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Just waiting for Mult2 (DIVD) to complete

Faster than light computation (skip a couple of cycles)

Tomasulo Example Cycle 55

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
55	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
0	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			

- Mult2 (DIVD) is completing; what is waiting for it?

Tomasulo Example Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	FU	M*F4	M(A2)		(M-M+M	(M-M)	Result			

- Once again: In-order issue, out-of-order execution and out-of-order completion.

Hardware-Based Speculation

- Exploiting instruction level parallelism requires to overcome the limitation of control dependence.
- Control dependence can be overcome by speculating on the outcome of branches and executing the program as if our guesses were correct.
- With speculation, we fetch, issue, and execute instructions as if our branch predictions were always correct.
- A mechanism is needed to handle the situation where speculation is incorrect.

Hardware-Based Speculation

- H/w based speculations combines three key ideas:
  - Dynamic branch prediction
    - To choose which instruction to execute
  - Speculation
    - To allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequences)
  - Dynamic scheduling
    - To deal with the scheduling of different combination of basic blocks.
- Dynamic scheduling without speculation only partially overlaps basic block because it requires that the branch be resolved before actually executing any instructions in the successor basic block.

Hardware-Based Speculation

- Execute instructions along predicted execution paths but only commit the results if prediction was correct
- Instruction commit: allowing an instruction to update the register file/meory when instruction is no longer speculative
- Need an additional piece of hardware to prevent any irrevocable action until an instruction commits
  - i.e. updating state or taking an execution

Reorder Buffer

- Reorder buffer – holds the result of instruction between completion and commit
  - Also used to pass results among the instructions that may be speculated

- Each entry in ROB contains Four fields:
  - Instruction type: branch/store/register
  - Destination field: register number or mem address
  - Value field: value of the instruction result until instruction commit
  - Ready field: indicates that the instruction has completed execution, and the value is ready.

- Modify reservation stations:
  - Operand source is now reorder buffer instead of functional unit

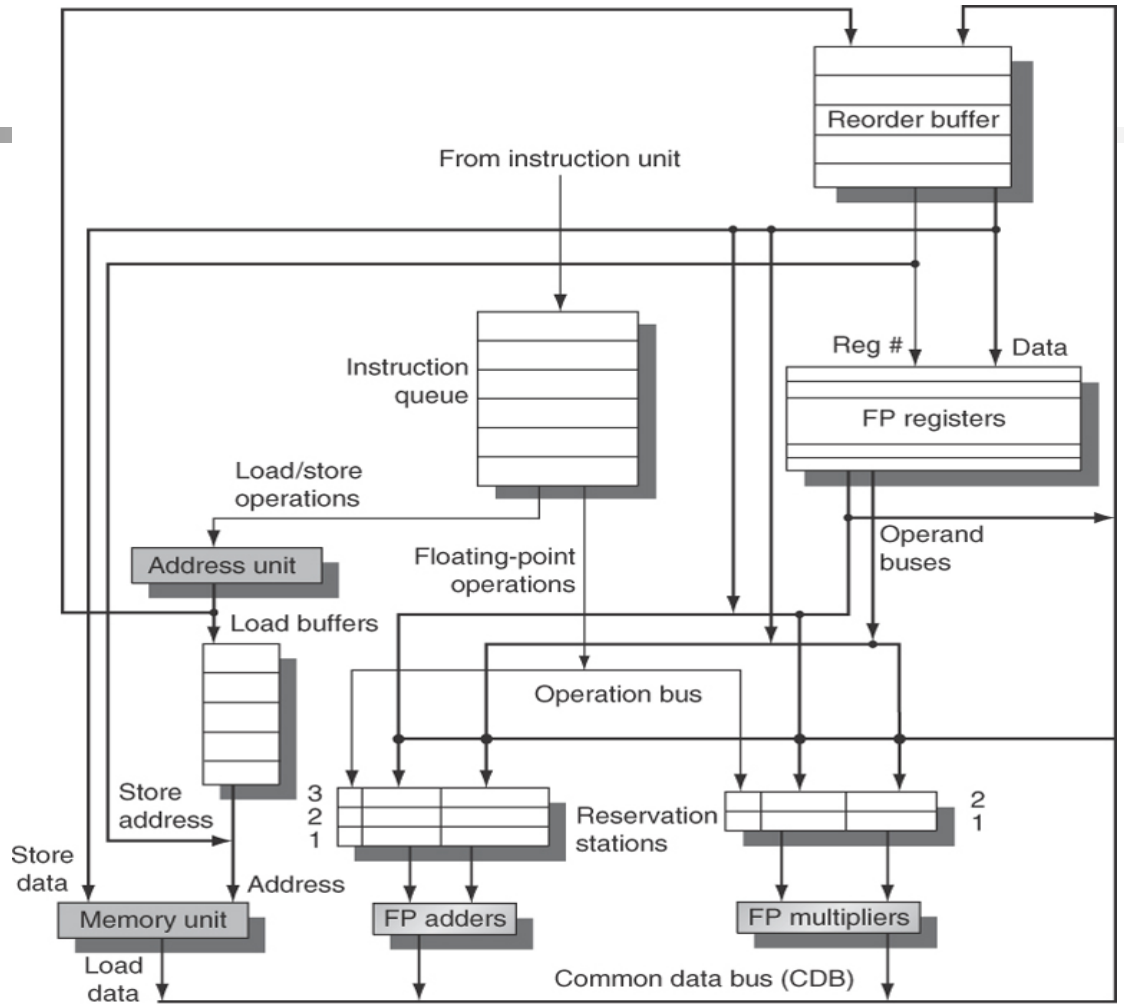


Figure 3.11 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure which implemented Tomasulo's algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB wider to allow for multiple completions per clock.

Reorder Buffer

- Register values and memory values are not written until an instruction commits
- On misprediction:
  - Speculated entries in ROB are cleared
- Exceptions:
  - Not recognized until it is ready to commit

Reorder Buffer

- Issue
  - Issue the instruction if there is an empty reservation station and an empty slot in the ROB
  - ROB number is used to TAG the result when it is placed on the CDB
- Execute
- Write result
- Commit
  - Three different sequence of action at commit depending on whether the committing instruction is a branch with an incorrect prediction, a store or any other instruction (normal commit)

Reorder Buffer

- Commit
  - Normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer
    - Processor updated the register with the result and removes the instruction from the ROB
  - Store memory is updated.
  - When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong.
    - The ROB is flushed and execution is restarted at the correct successor of the branch.

Multiple Issue and Static Scheduling

- To achieve CPI < 1, need to complete multiple instructions per clock

- Solutions:
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors

Multiple Issue Processor

- Statically scheduled superscalar processors
  - Issue varying number of instruction / CC
  - In-order execution
- Dynamically scheduled superscalar processors
  - Issue varying number of instruction / CC
  - Out-order execution
  - VLIW (very long instruction word) processors
  - Package multiple operations into one instruction

Multiple Issue

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

VLIW Processors

- VLIWs uses multiple independent functional units

- To keep FU busy there must be enough parallelism in code to fill the available operation slots

VLIW Processors

- Disadvantages:
  - Statically finding parallelism
  - Code size
  - No hazard detection hardware
    - Early VLIWs operated in lockstep
  - Binary code compatibility
    - In VLIW approach, the code sequence make use of both the instruction set definition and the detailed pipeline structure (including FU and their latencies) .

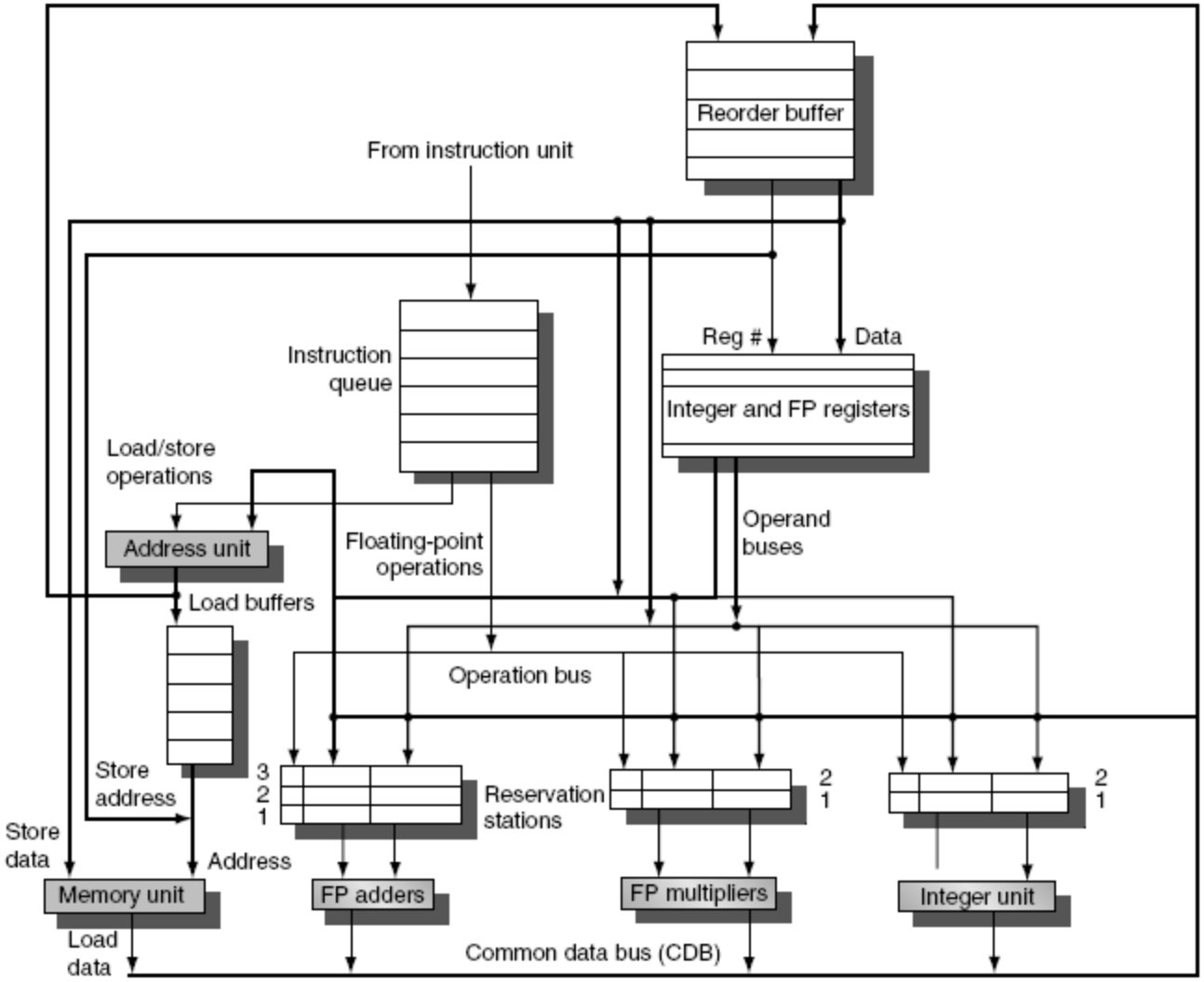
Dynamic Scheduling, Multiple Issue, and Speculation

- Modern microarchitectures:
  - Dynamic scheduling + multiple issue + speculation
- Two approaches have been used to issue multiple instruction per clock, and both rely on the observation that:
  - The key is assigning a reservation and updating the pipeline control tables

Dynamic Scheduling, Multiple Issue, and Speculation

- Two approaches:
  - Assign reservation stations and update pipeline contro table in half clock cycles
    - Only supports 2 instructions/clock
  - Build the logic necessary to handle two or more instructions at once, including any possible dependencies between the instructions
  - Hybrid approaches
- Issue logic can become bottleneck

Overview of Design



Multiple Issue

- Strategy for updating the issue logic and the reservation table in a dynamically scheduled superscalar with up to n issues per clock:
  - Limit the number of instructions of a given class that can be issued in a “bundle”
    - i.e. on FP, one integer, one load, one store
  - Examine all the dependencies among the instructions in the issue bundle
  - If dependencies exist in bundle, encode them in reservation stations

Example

```
Loop:  LD R2,0(R1)        ;R2=array element
       DADDIU R2,R2,#1    ;increment R2
       SD R2,0(R1)        ;store result
       DADDIU R1,R1,#8    ;increment pointer
       BNE R2,R3,LOOP     ;branch if not last element
```

Consider the execution of the following loop, which increments each elements of an integer array on a dual-issue processor (with/with out speculation). Assuming that there are separate FU for effective address calculation, ALU operations, and branch condition evaluation. Create a table for the first 3 iteration of this loop for both processor. Two instructions of any type can commit per cycle.

Example (No Speculation)

Iteration number	Instructions		Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Execute directly
1	BNE	R2,R3,LOOP	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#8	5	8		9	Wait for BNE
2	BNE	R2,R3,LOOP	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#8	8	14		15	Wait for BNE
3	BNE	R2,R3,LOOP	9	19			Wait for DADDIU

Example

Iteration number	Instructions		Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	LD	R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU	R2,R2,#1	1	5		6	7	Wait for LW
1	SD	R2,0(R1)	2	3			7	Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	8	Commit in order
1	BNE	R2,R3,LOOP	3	7			8	Wait for DADDIU
2	LD	R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU	R2,R2,#1	4	8		9	10	Wait for LW
2	SD	R2,0(R1)	5	6			10	Wait for DADDIU
2	DADDIU	R1,R1,#8	5	6		7	11	Commit in order
2	BNE	R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LD	R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU	R2,R2,#1	7	11		12	13	Wait for LW
3	SD	R2,0(R1)	8	9			13	Wait for DADDIU
3	DADDIU	R1,R1,#8	8	9		10	14	Executes earlier
3	BNE	R2,R3,LOOP	9	13			14	Wait for DADDIU

Advanced Techniques for Instruction Delivery and Speculation

- In a high performance pipeline, especially one with multiple issue, predicting branches well is not enough
  - Deliver high instruction bandwidth stream
  - Delivering 4 – 8 instruction in every clock cycle
- For high instruction delivery
  - A branch target buffer
  - An integrated IF unit
  - Dealing with indirect branches by predicting return addresses

Branch-Target Buffers

- Branch-prediction cache that stores the predicted address for the next instruction after a branch:
  - A branch prediction buffer is accessed during the ID cycle, so at the end of the ID we know the branch target address. Thus, by the end of the ID we know enough to fetch the next predicted instruction.
  - Accessing the target buffer during the IF stage using the instruction address of the fetched instruction (a possible branch) to index the buffer.

Branch-Target Buffers

- Branch-prediction cache that stores the predicted address for the next instruction after a branch:
  - For a hit, the predicted instruction address is know at the ~~of the~~ IF cycle which is one cycle earlier than the branch prediction buffer.
  - Predicting the next instruction address before decoding the current instruction!, we must know whether the fetched instruction is predicted as a taken branch instruction

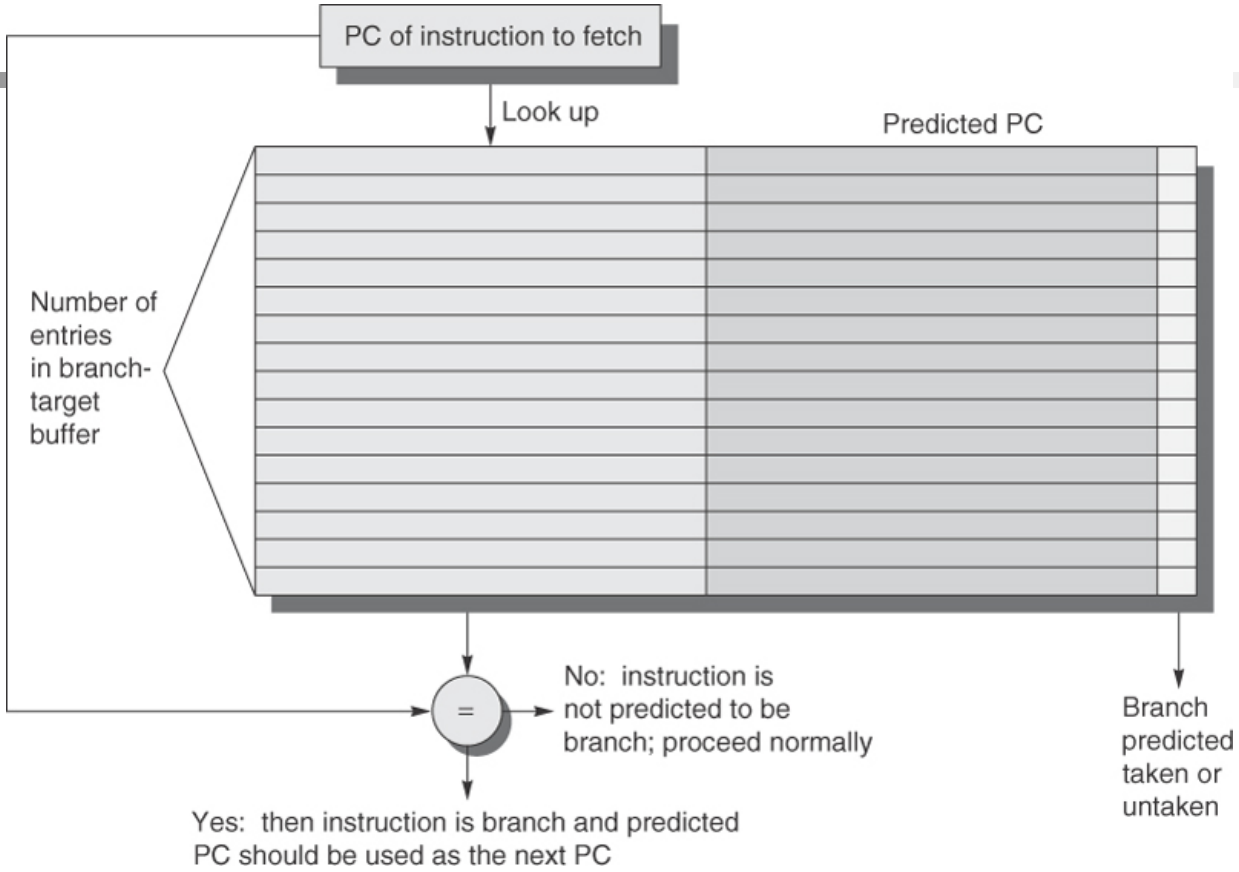


Figure 3.21 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.

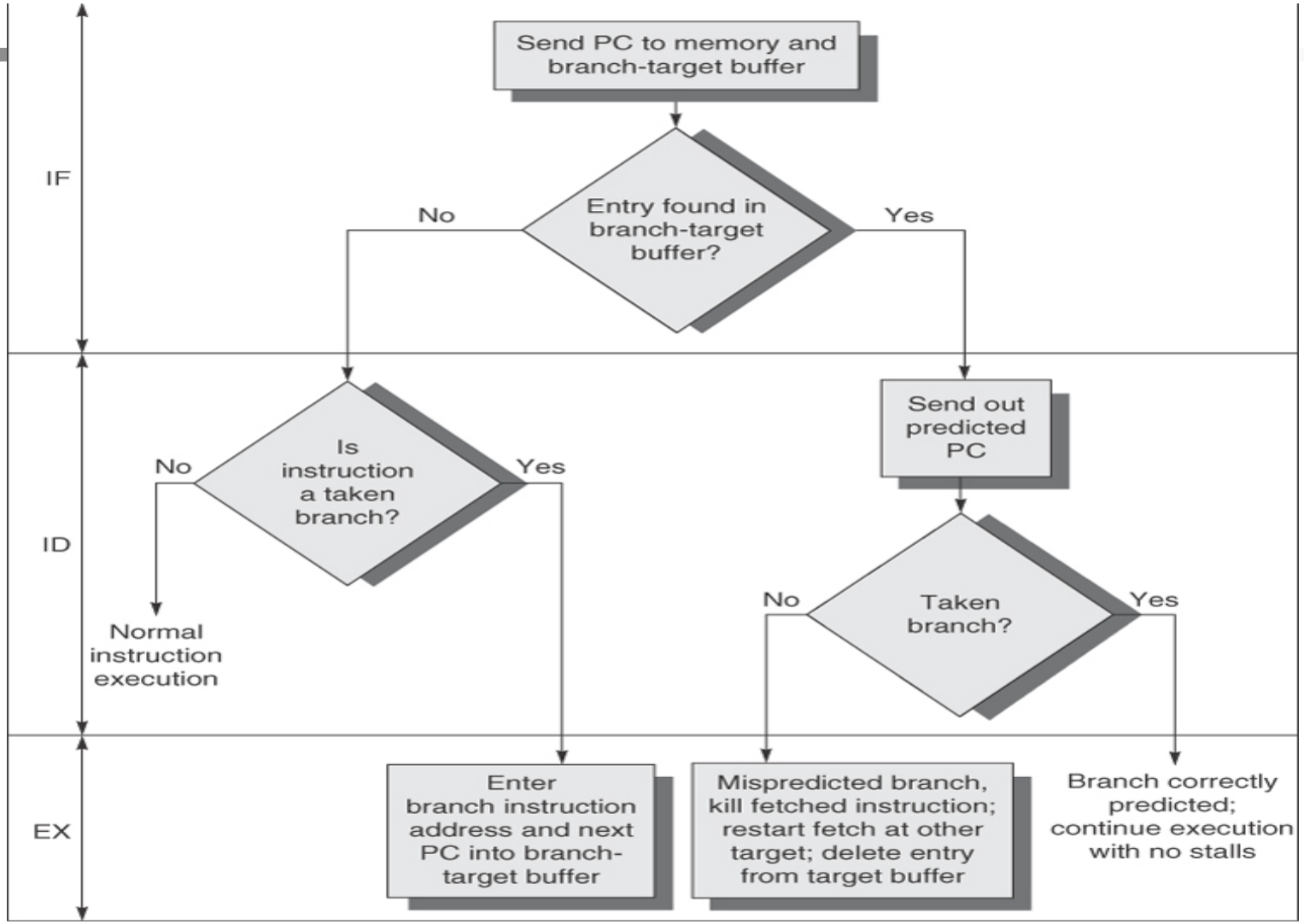


Figure 3.22 The steps involved in handling an instruction with a branch-target buffer.

Instruction in Buffer	Prediction	Actual branch	Penalty cycle
Yes	Taken	Taken	0
Yes	Taken	Not Taken	2
No		Taken	2
No		Not Taken	0

Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer

Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions as shown in the previous table. Make the following assumptions about the prediction accuracy and hit rate: Prediction accuracy is 90% (for instruction in the buffer; Hit rate in the buffer is 90% (for branches predicted taken)

- Pr(branch in buffer, but actually no taken
  = Percent buffer hit rate x Percent incorrect predictions
  = 90% x 10% = 0.09

- Pr(branch not in buffer, but actually taken) = 10%

- Branch Penalty = (0.09 + 0.10) x 2 = 0.38

Branch Target Buffer

- One variation of Branch target buffer is to store one or more target instruction instead of or in addition to, the predicted target address.
- Advantages
  - It allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer.
  - Buffering the actual target instructions allow us to perform an optimization called *branch folding*
- Branch folding can be used to obtain zero-cycle unconditional branch and some times zero-cycle conditional branch

Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack

Integrated Instruction Fetch Units

- To meet the demands of multiple-issue processors, recent designers have chosen to implement an integrated instruction fetch unit as a separate autonomous unit that feeds instructions to the rest of the pipeline.

- Integrated instruction fetch unit that integrates several functions:

  - Integrated branch prediction – The branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so as to drive the fetch pipeline

  - Instruction prefetch – To deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead, autonomously managing the prefetching of instructions and integrating it with branch prediction

Integrated Instruction Fetch Units

- Integrated instruction fetch unit that integrates several functions:
  - Instruction memory access and buffering – Encapsulates the complexity of fetching multiple instructions per clock, trying to hide the cost of crossing cache blocks, and provides buffering, acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed

Multithreading

- Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput
  - Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion
  - Multithreading shares most of the processor core among a set of threads
    - Duplicating only private state such as registers and program counter

Multithreading ….

- Hardware approach to Multithreading
  - Fine-Grained Multithreading
  - Coarse-Grained Multithreading
  - Simultaneous Multithreading

Multithreading ….

- Fine-Grained Multithreading
  - Switches between threads on each clock, causing the execution of instructions from multiple threads to be interleaved
  - Interleaving is often done in round robin fashion, skipping any thread that are stopped at that time
- Advantage:
  - Hide the throughput losses that arise from both short and long stall
- Disadvantage:
  - Slows down the execution of an individual thread, since a thread that is ready to execute without stall will be delayed by instructions from other thread
  - Coarse-Grained Multithreading
  - Simultaneous Multithreading

Multithreading ….

- Coarse-Grained Multithreading
  - Switches threads only on costly stalls, such as level two or three cache misses each clock, causing the execution of instructions from multiple threads to be interleaved
  - Interleaving is often done in round robin fashion, skipping any thread that are stopped at that time
- Advantage:
  - Hide the throughput losses that arise from both short and long stall
- Disadvantage:
  - Slows down the execution of an individual thread, since a thread that is ready to execute without stall will be delayed by instructions from other thread

  - Coarse-Grained Multithreading

  - Simultaneous Multithreading

- Enroll Using

  Class ID: 12429332
  Enrollment Password: aca2016

  Must Have Turnitin Account

ARM Cortex-A8

- Apple ipad, Motorola Droid, iPhones 3GS and 4
- A8 is a dual issue, statically scheduled superscalar with dynamic issue detection
- 13-stage pipeline
- Dynamic branch predictor
  - with a 512-entry two-way set associative branch target buffer
  - 4K-entry global history buffer (index by the branch history and the current PC)
- In the event that the branch target buffer misses, a prediction is obtained from the global history buffer, which can then be used to compute the branch address

ARM Cortex-A8

- Eight-entry return stack is kept to track the return address
- Incorrect prediction results in a 13-cycle penalty as the pipeline is flushed.

ARM Cortex-A8

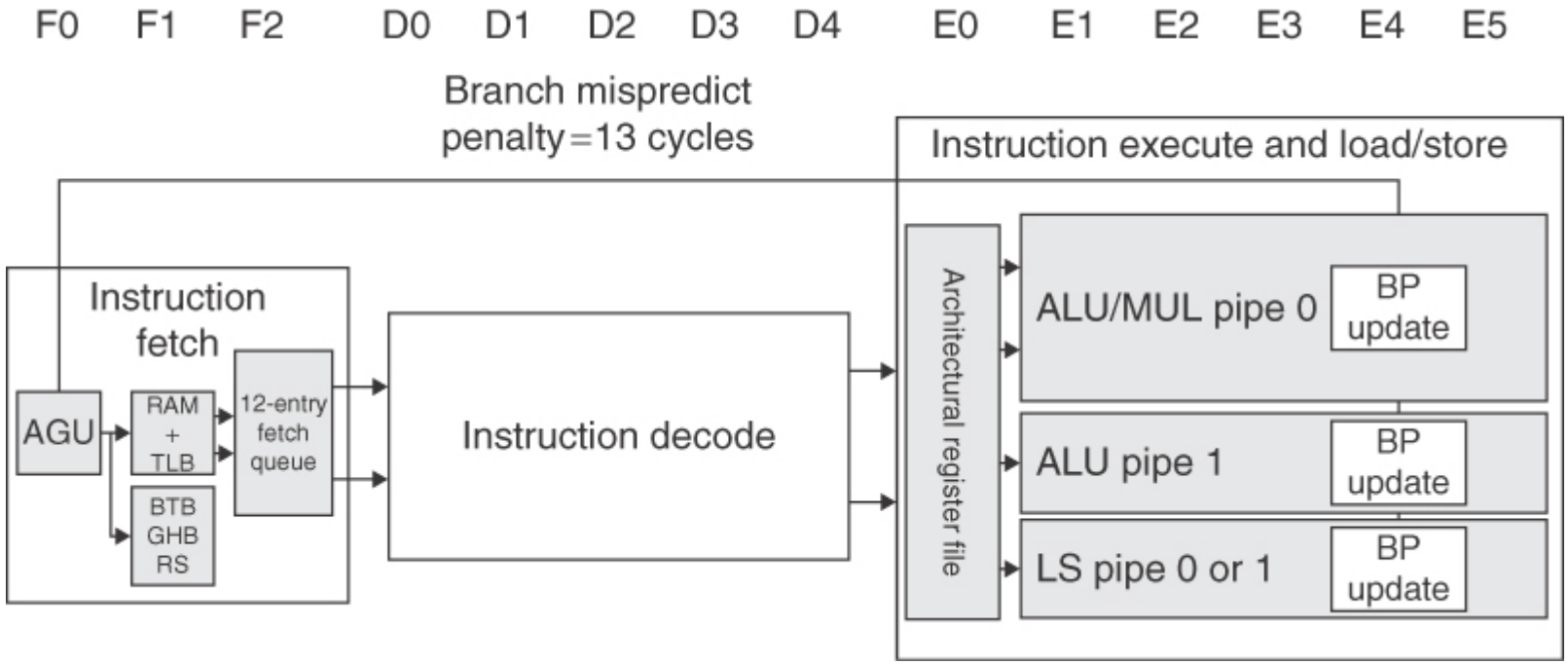


Figure 3.36 The basic structure of the A8 pipeline is 13 stages. Two cycles are used for instruction fetch and five for instruction decode, in addition to a six-cycle integer pipeline. This yields a 13-cycle branch misprediction penalty. The instruction fetch unit tries to keep the 12-entry instruction queue filled.

ARM Cortex-A8 – Decoder

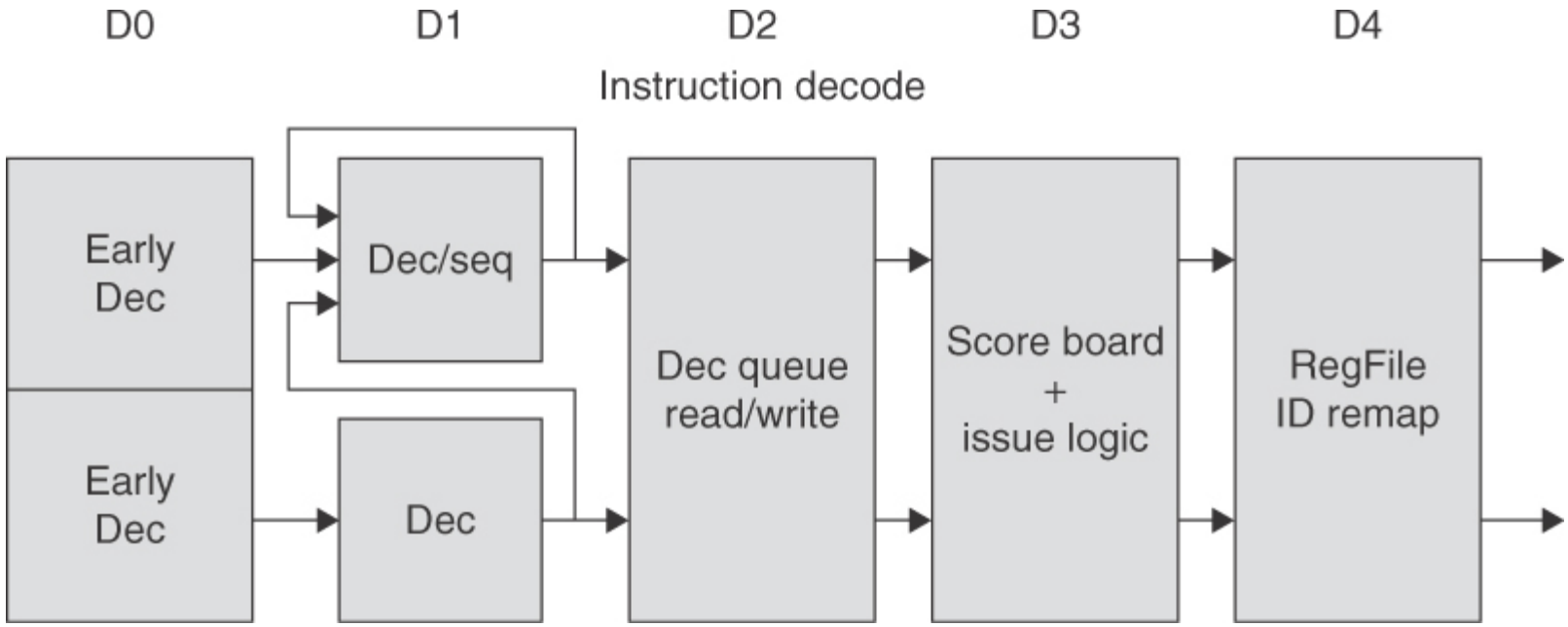


Figure 3.37 The five-stage instruction decode of the A8. In the first stage, a PC produced by the fetch unit (either from the branch target buffer or the PC incrementer) is used to retrieve an 8-byte block from the cache. Up to two instructions are decoded and placed into the decode queue; if neither instruction is a branch, the PC is incremented for the next fetch. Once in the decode queue, the scoreboard logic decides when the instructions can issue. In the issue, the register operands are read; recall that in a simple scoreboard, the operands always come from the registers. The register operands and opcode are sent to the instruction execution portion of the pipeline.

ARM Cortex-A8 – Execution

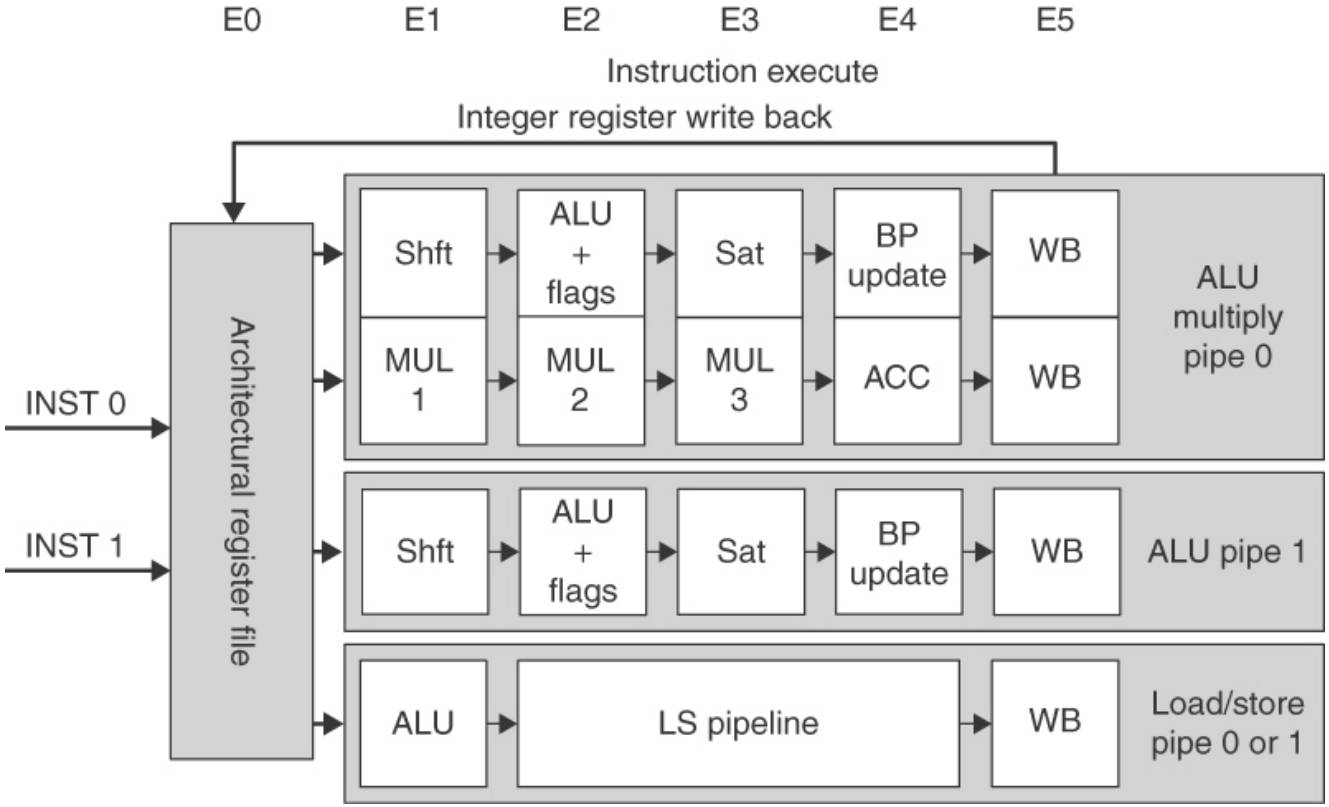


Figure 3.38 The six-stage execution of the A8. Multiply operations are always performed in ALU pipeline 0. Either instruction 1 or instruction 2 can go to the load/store pipeline

Performance of A8 Pipeline

- A8 has an ideal CPI of 0.5
- Three sources of pipeline stall
  - Functional hazards
    - To adjacent instruction selected for issue simultaneously use the same functional pipeline
  - Data hazards
    - Stall both or the second of a pair
  - Control hazards
    - When branches are mis-predicted

Memory Hierarchy A8

- Supports Two-level cache hierarchy
  - First level a pair of caches (I and D)
  - Each 16/32 KB organized as four-way set associative
  - Way prediction
  - Random replacement
  - Second level (optional)
  - 8-way set associative
  - 128 KB to 1MB
  - Organized into one to four banks to allow several transfers from memory to occur simultaneously
  - External bus of 64 to 128 bits to handle memory request

Memory Hierarchy A8

- First-level cache is virtually indexed and physically tagged
- Second-level is physically indexed and tagged
- Both level uses 64-byte block size
- For a D-cache of 32 KB and a page size of 4 KB, each physical page could map to two different cache address

Memory Hierarchy A8

- Memory management is handled by a pair of TLBs (I and D)
- Fully associative, 32 entries
- Variable page size (4 KB, 16 KB, 64 KB, 1 MB, and 16 MB)
- TLB replacement is done by round robin algorithm
- 32-bit virtual address is used to index the TLB and the caches, assuming 32 KB primary caches and a 512 KB secondary caches

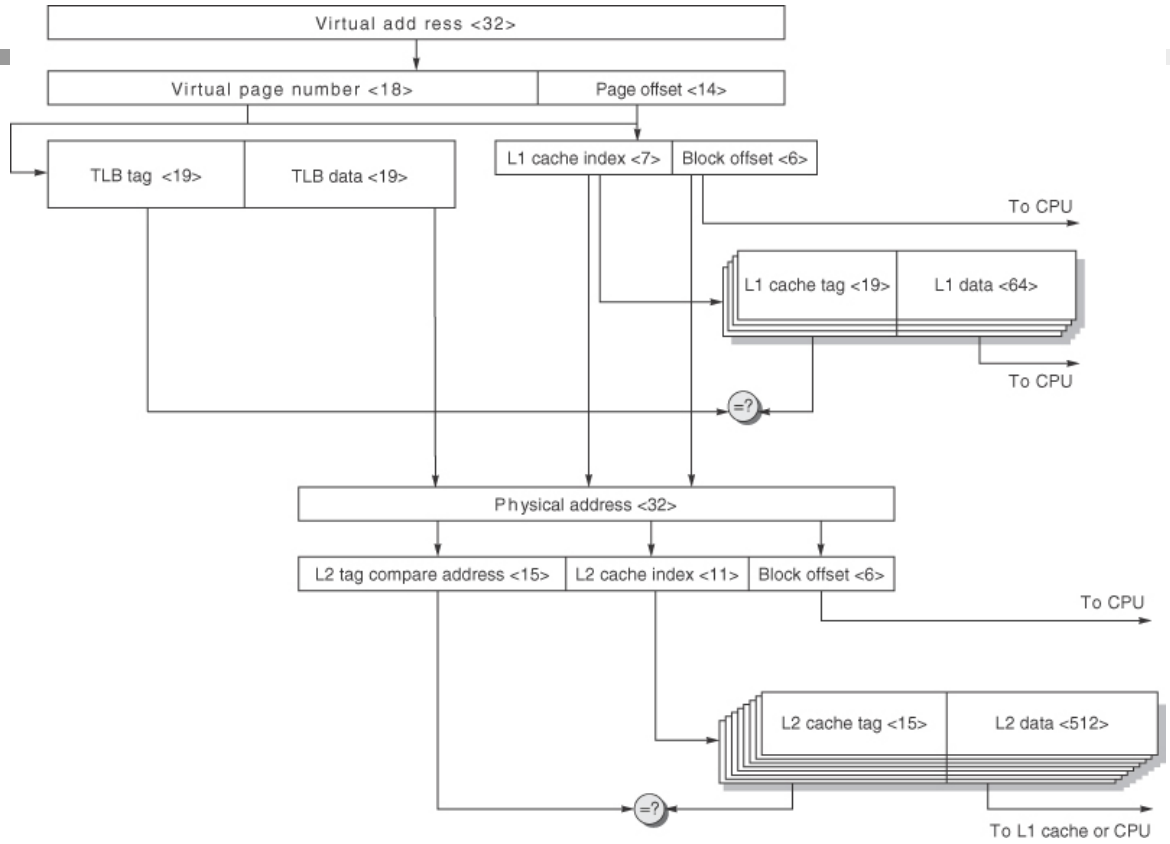


Figure 2.16 The virtual address, physical address, indexes, tags, and data blocks for the ARM Cortex-A8 data caches and data TLB. Since the instruction and data hierarchies are symmetric, we show only one. The TLB (instruction or data) is fully associative with 32 entries. The L1 cache is four-way set associative with 64-byte blocks and 32 KB capacity. The L2 cache is eight-way set associative with 64-byte blocks and 1 MB capacity. This figure doesn't show the valid bits and protection bits for the caches and TLB, nor the use of the way prediction bits that would dictate the predicted bank of the L1 cache.

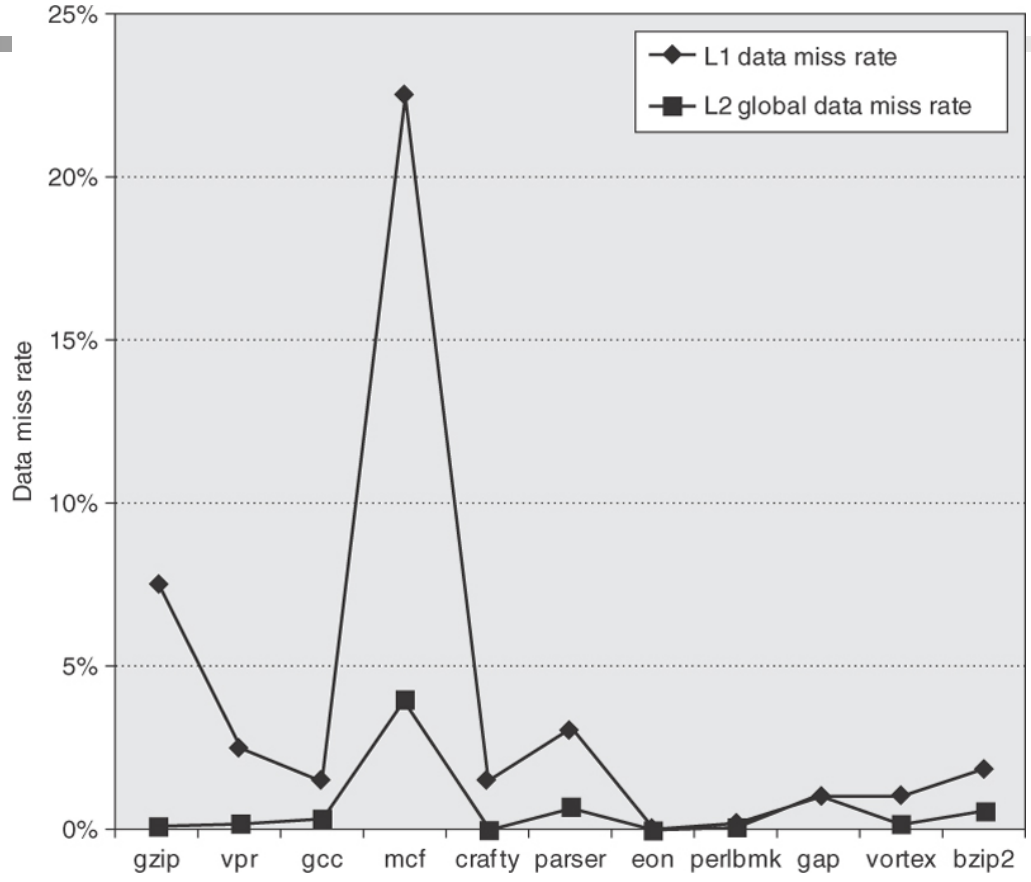


Figure 2.17 The data miss rate for ARM with a 32 KB L1 and the global data miss rate for a 1 MB L2 using the integer Minnespec benchmarks are significantly affected by the applications. Applications with larger memory footprints tend to have higher miss rates in both L1 and L2. Note that the L2 rate is the global miss rate, that is counting all references, including those that hit in L1. Mcf is known as a cache buster.

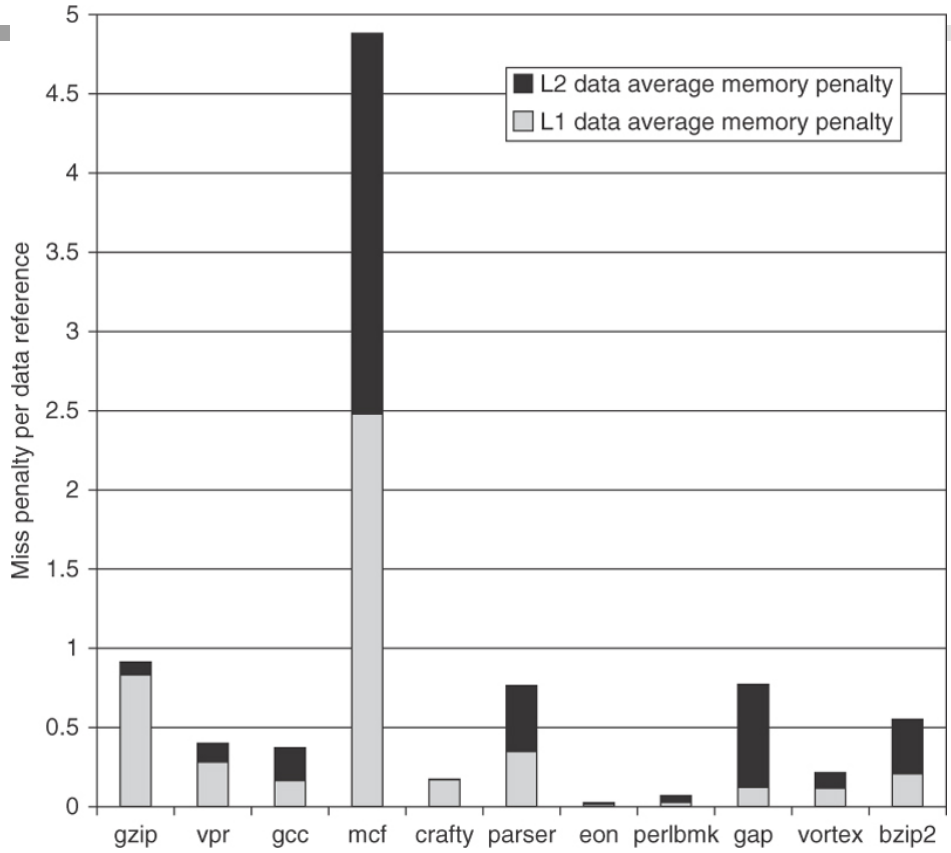


Figure 2.18 The average memory access penalty per data memory reference coming from L1 and L2 is shown for the ARM processor when running Minniespec. Although the miss rates for L1 are significantly higher, the L2 miss penalty, which is more than five times higher, means that the L2 misses can contribute significantly.

Intel Core i7

- Dynamically scheduled speculative processor
- 14-stage pipeline
- 4-issue per CC
- IF
  - Uses multilevel branch target buffer to achieve a balance between speed and prediction accuracy
  - Return stack address to speed up return function
  - Misprediction causes a penalty of 15 CC
  - IF unit fetches 16 bytes from the I-Cache, using the predicted address

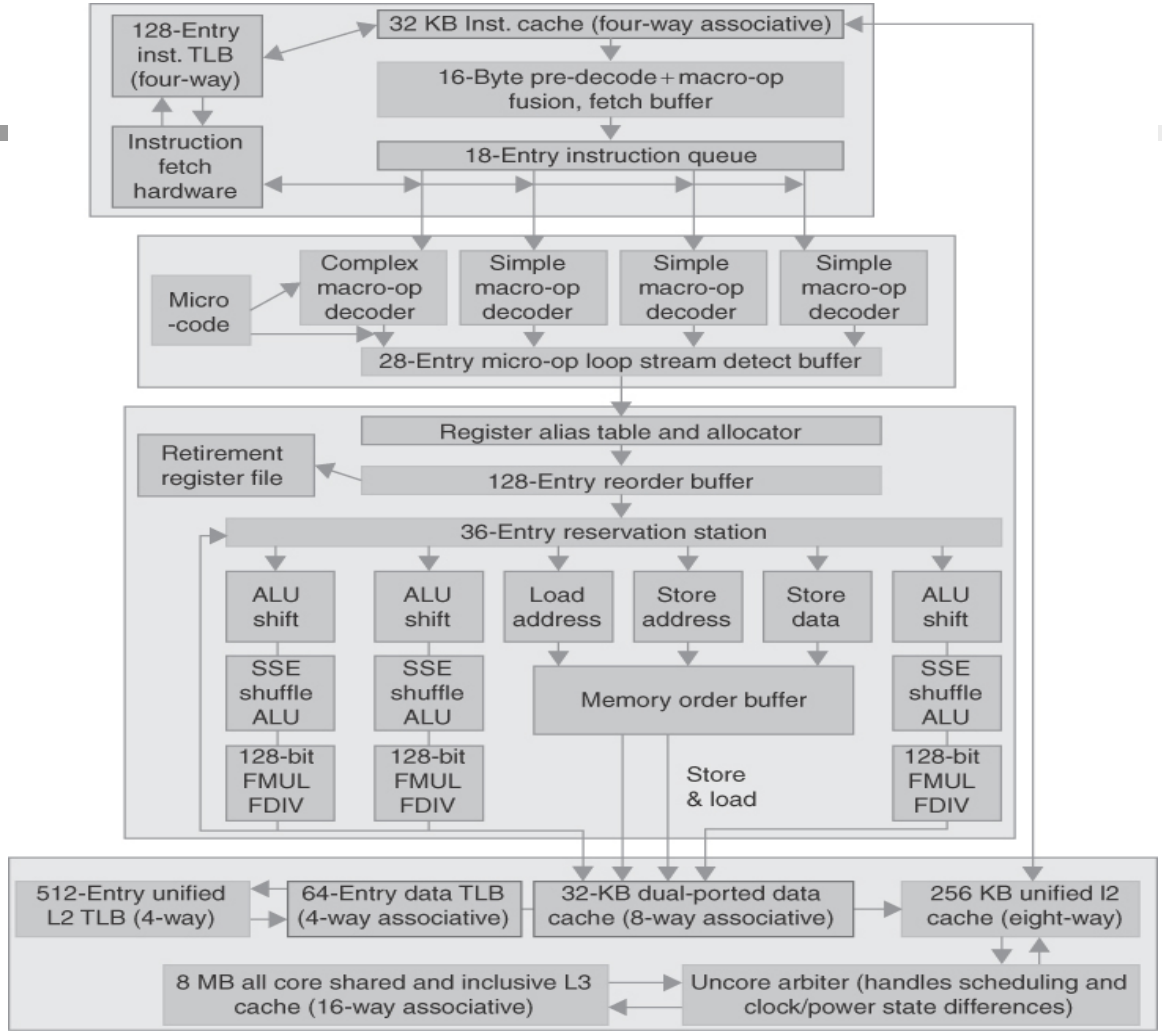


Figure 3.41 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions costing 17 cycles. There are 48 load and 32 store buffers. The six independent functional units can each begin execution of a ready micro-op in the same cycle.

Intel Core i7

- Predecode stage buffer
  - 16 bytes fetched are placed in the predecode instruction buffer
  - A process called macro-op fusion is executed
  - Macro-op fusion takes instruction combinations such as compare followed by a branch and fuses them into a single operation
  - Breaks the 16 bytes into individual x86 instructions
  - Individual x86 instructions are placed into the 18-entry instruction queue

Intel Core i7

- Micro-op decode
  - Individual x86 instructions are translated into micro-ops
  - Micro-ops are placed in the 28-entry micro-op buffer
- Micro-op buffer
  - Performs loop-stream detection and microfusion
  - Loop-stream detector will find loop and directly issue the micro-ops from the buffer
  - Microfusion combines instruction pairs such as load/ALU operation and ALU operation/store and issues them to a single reservation station , increasing the usage of the buffer

Memory Hierarchy

- 48-bit virtual address and 36-bit physical address
- Memory management is handled with a two level TLB
- Three-level cache hierarchy
  - First-level caches are virtually indexed and physically tagged
  - L2 and L3 caches physically indexed

i7 TLB Structure

Characteristic	Instruction TLB	Data TLB	Second-Level TLB
Size	128	64	512
Associativity	4-way	4-way	4-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU
Access Latency	1 cycle	1 cycle	6 cycle
Miss	7 cycle	7 cycle	Hundreds of cycle to access page table

i7 3-level cache hierarchy

Characteristic	L1	L2	L3
Size	32 KB (I,D)	256 KB	2 MB per Core
Associativity	4-way/8-way	8-way	16-way
Replacement	Pseudo-LRU	Pseudo-LRU	Pseudo-LRU but with ordered selection algithm
Access Latency	4 Cycle, Pipelined	10 cycle	35 cycle

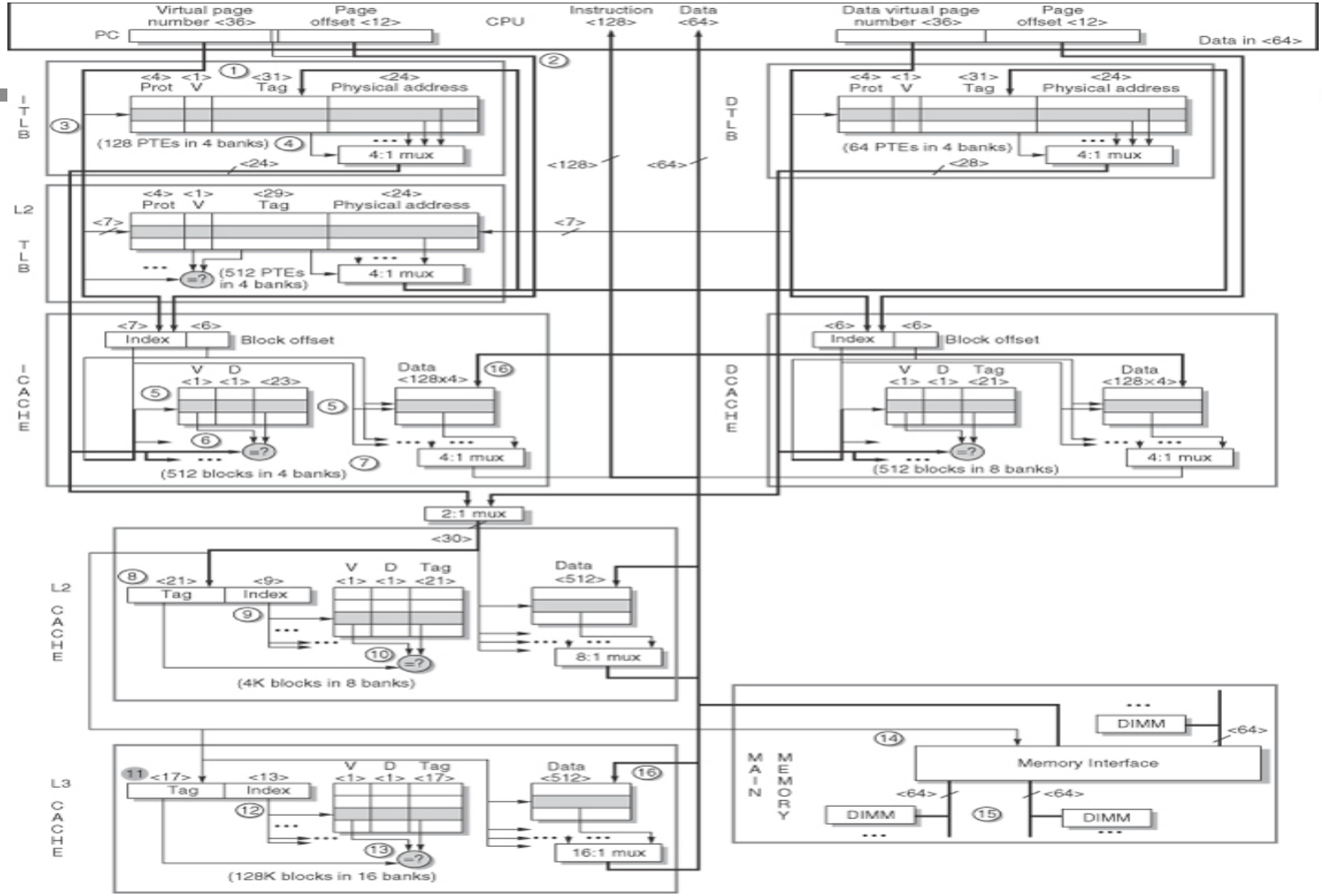


Figure 2.21 The Intel i7 memory hierarchy and the steps in both instruction and data access. We show only reads for data. Writes are similar, in that they begin with a read (since caches are write back). Misses are handled by simply placing the data in a write buffer, since the L1 cache is not write allocated.